ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

GENERADOR DE CONJUNTOS DE PRUEBA PARAMÉTRICOS
BPELUnit PARA COMPOSICIONES WS-BPEL

Jose Luis Ezquerro Casado

21 de junio de 2013

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

GENERADOR DE CONJUNTOS DE PRUEBA PARAMÉTRICOS BPELUnit PARA COMPOSICIONES WS-BPEL

DEPARTAMENTO: INGENIERÍA INFORMÁTICA

DIRECTORES DEL PROYECTO: JUAN JOSÉ DOMÍNGUEZ JIMÉNEZ Y ANTONIO GARCÍA DOMÍNGUEZ

AUTOR DEL PROYECTO: JOSE LUIS EZQUERRO CASADO

Cádiz, 21 de junio de 2013

Fdo: Jose Luis Ezquerro Casado

Licencia

Agradecimientos

- A mi familia, mi pareja y su encantadora hija, por apoyarme y animarme en estos meses de horas de trabajo y esfuerzo.
- Juan José Domínguez Jiménez, por hablarme del grupo de investigación UCASE y darme la oportunidad de ser alumno colaborador.
- A Antonio García Domínguez, por sus grandes enseñanzas en los seminarios y su enorme apoyo al grupo.
- Al resto de miembros del grupo UCASE, por su ayuda e ideas.

Notación y formato

En esta sección incluiremos los aspectos relevantes a la notación y el formato a lo largo del documento. Dicha notación es la siguiente: Si nos referimos a un directorio en particular, usaremos la notación:

/home

Cuando nos referimos a un fichero de un tipo en concreto (por ejemplo un fichero con extensión "bpts"), utilizaremos la notación:

bpts

Cuando nos referimos a un comando, o función de un lenguaje, usaremos la notación:

`clientTrack`

Cuando nos referimos a una clase, utilizaremos la notación:

MiClase

Cuando nos referimos a una tecnología o herramienta, lo indicaremos con el nombre propio:

BPELUnit, TestGenerator, etc.

Índice general

Índice de figuras

Capítulo 1

Motivación y contexto

1.1. Introducción

El presente Proyecto Fin de Carrera (PFC) se enmarca como trabajo de colaboración dentro del grupo de investigación "UCASE de Ingeniería del Software" de la Universidad de Cádiz. La actividad de este grupo comprende actualmente las áreas de ingeniería de servicios, arquitectura dirigidas por eventos, arquitecturas orientadas a servicios (SOA), desarrollo dirigido por modelos, prueba de software y verificación, y validación de software.

Dentro de la línea de investigación *Prueba de Software y Verificación* se trabaja en la mejora de conjuntos de casos de prueba para composiciones de Servicios Web (WS o Web Services en inglés) escritas en WS-BPEL (Web Services Business Process Execution Language) 2.0 [22].

El lenguaje WS-BPEL es un lenguaje basado en XML, estandarizado por OASIS para componer WS. Permite especificar la lógica de la composición de los servicios (envío de mensajes, sincronización, iteración, tratamiento de transacciones erróneas, etc.) independientemente de su implementación [7]. Con este lenguaje se puede integrar y automatizar los procesos de negocio usando los estándares de los WS, y al estar basado en XML es portable y se puede ejecutar en cualquier motor WS-BPEL.

Los WS permiten el desarrollo de aplicaciones distribuidas de manera rápida, simple y con coste bajo, motivo por el cual están cobrando protagonismo a la hora de definir

procesos de negocio. Por lo tanto, la prueba de este tipo de software se torna esencial.

Gran parte de las investigaciones del grupo se centran en las pruebas de mutaciones a composiciones WS-BPEL. La prueba de mutaciones es una técnica de pruebas de caja blanca que consiste en introducir pequeños cambios sintácticos en el programa original, creando un nuevo programa (un *mutante*[1]) por cada cambio. Sirve para evaluar las pruebas, viendo si pueden diferenciar al programa original del mutante.

Una vez se tienen los mutantes, se ejecuta al programa original y a los mutantes el mismo conjunto de casos de prueba, recogidos en un fichero *bpts* (BPELUnit Test Suite), para, posteriormente, poder estudiar los resultados. BPELUnit [18] es un marco de pruebas unitarias automáticas, repetibles y de caja blanca para probar composiciones WS-BPEL.

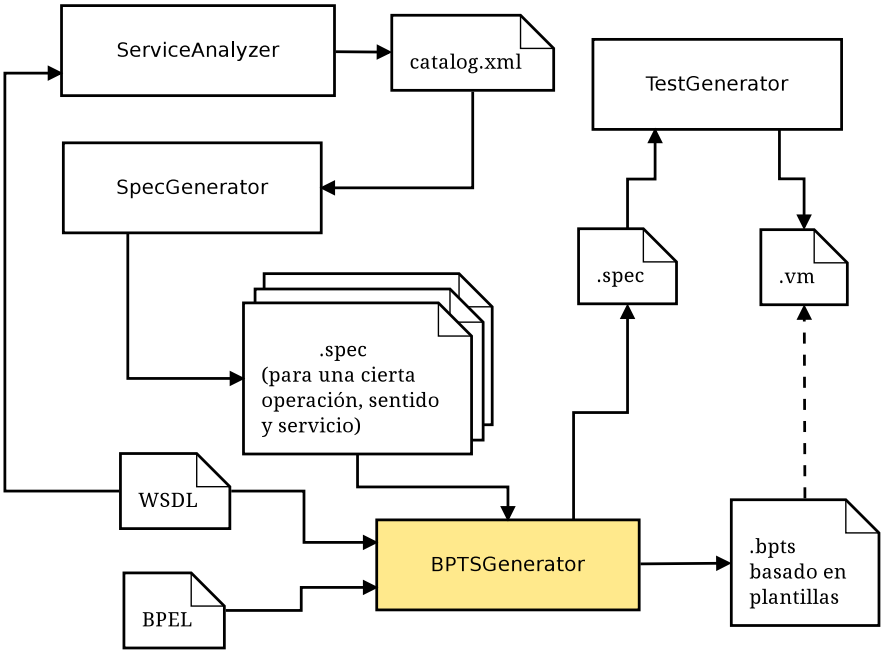

Figura 1.1: Visión general del proyecto

El conjunto de casos de prueba para BPELUnit (fichero *bpts*) es un fichero XML que define qué proceso se va a probar y cómo. BPELUnit puede construir el cuerpo de los

[1]Los mutantes que contienen una sola diferencia respecto al programa original son de orden uno o "mutantes de primer orden". Si se realiza más de un cambio sintáctico al programa original se obtiene un "mutante de orden superior".

mensajes usando plantillas Velocity [10], que permiten al usuario definir fácilmente muchos casos de prueba con actividades parecidas pero que tienen diferente contenido en sus mensajes [17]. Estas plantillas permiten generar los mensajes de cada caso de prueba a partir de una serie de variables predefinidas, alojadas en un fichero de extensión "vm". De la producción de las plantillas Velocity se ocupa una herramienta llamada ServiceAnalyzer [14] creada por Cristina Jiménez Gavilán. Esta herramienta se encarga del análisis de los Servicios Web que intervienen en la composición.

Estos ficheros *vm* pueden hacerse a mano, dando los valores a las variables predefinidas que el programador desee; o bien se puede crear de una manera aleatoria gracias a una herramienta creada por Miguel Ángel Pérez Montero, llamada TestGenerator [24]. Esta herramienta es capaz, entre otras cosas, de generar este fichero *vm* a partir de otro con extensión "spec", en el cual se declaran los tipos de datos y las variables que se utilizarán en las plantillas Velocity del *bpts*. Gracias a otra herramienta creada por el grupo UCASE llamada SpecGenerator (integrada en TestGenerator) se puede crear un fichero *spec* de cada operación de cada servicio (ya que en un mismo servicio pueden definirse varias operaciones).

Hasta ahora solamente se podían crear los ficheros *bpts* a mano, con lo cual había que tener cierto dominio en BPELUnit, saber crear plantillas Velocity, etc. De esta manera, se propuso crear un proyecto que consiguiera abstraer estas tecnologías al programador y poder generar el fichero *bpts* junto con un catálogo de pruebas (fichero *vm*) para una composición.

1.2. Objetivos

Con la realización de este proyecto se pretende crear una herramienta que pueda analizar composiciones WS-BPEL para posteriormente poder generar el fichero *bpts* con el que poder realizar pruebas paramétricas[2] BPELUnit a la composición analizada, junto con una colección de datos de entrada aleatorios.

Para llevar a cabo el objetivo principal del proyecto, es necesario:

[2] Las pruebas paramétricas permiten, mediante una misma plantilla (en este caso Apache Velocity), construir mensajes con el mismo formato pero distinto contenido a través del uso de variables. [14]

- Realizar un análisis de la composición para extraer la información necesaria de los servicios que intervienen en la misma.
- Extraer las plantillas Velocity de cada servicio por medio de ServiceAnalyzer.
- Crear el fichero *bpts* gracias al framework de BPELUnit.
- Almacenar en un único *spec* todos los tipos y todas las variables de los servicios analizados de la composición.
- Generar el fichero *vm*, con una colección de datos de entrada aleatorios.

1.3. Alcance

Este proyecto trabajará con el sistema de tipos de las composiciones WS-BPEL que actualmente se están estudiando, de forma que cubre las necesidades del grupo UCASE existentes por el momento.

Este proyecto conseguirá integrar en una única herramienta los proyectos Service-Analyzer, TestGenerator y SpecGenerator, cumpliendo así el objetivo de analizar una composición WS-BPEL con sus respectivos servicios y generar un catálogo de pruebas.

Por limitaciones de tiempo, se ha realizado un primer análisis de la lógica de la composición, restringiendo en la composición WS-BPEL a que si se desea llamar un mismo servicio dos veces ha de hacerse con variables distintas.

La comunicación entre el usuario y la herramienta se hará a través de la línea de órdenes.

1.4. Conceptos básicos

1.4.1. Acrónimos

ASL Apache Software License

AST Abstract Syntax Tree

BPEL Business Process Execution Language

BPTS BPELUnit Test Suite

CASE Computer Aided Software Engineering

CI Continuous Integration

CPL Common Public License

CSV Comma-Separated Values

DOM Document Object Model

DTD Document Type Definition

EPL Eclipse Public License

GNU GNU is Not Unix

GPL GNU General Public License

HTML Hyper Text Markup Language

HTTP HyperText Transfer Protocol

IEEE Institute of Electrical and Electronics Engineers

J2SE Java 2 Standard Edition

MOJO Maven Old Java Object

OASIS Organization for the Advancement of Structured Information Standards

POM Project Object Model

SCV Sistema de Control de Versiones

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

UML Unified Modeling Language

VTL Velocity Template Language

W3C World Wide Web Consortium

WS Web Services

WS-BPEL Web Services Business Process Execution Language

WSDL Web Services Description Language

WWW World Wide Web

XSD XML Schema Definition

XML eXtensible Markup Language

XPath XML Path Language

XQuery XML Query

XHTML eXtensible Hyper Text Markup Language

XSLT eXtensible Stylesheet Language Transformations

XSL eXtensible Stylesheet Language

1.4.2. Definiciones

Mockup Servicio sustituido de un servicio real en una simulación, que implementa un comportamiento predefinido, y que suele emplearse en pruebas para reemplazar a un servicio costoso, al que no se puede acceder o también para comprobar si la salida de un programa para un caso de prueba es la esperada. No tienen lógica interna, limitándose a responder con mensajes predefinidos, o "fallando" si así se especifica.

Proceso de negocio síncrono Proceso de negocio en el que el cliente queda a la espera de su respuesta sin cortar la comunicación.

Proceso de negocio asíncrono Proceso de negocio en el que el cliente continúa su ejecución tras enviar la petición, y el proceso servidor le enviará posteriormente una notificación del resultado de la petición.

Orquestación La orquestación de Servicios Web se basa en un modelo centralizado en el cual las interacciones no se realizan directamente entre los Servicios Web sino que existe una entidad encargada de definir la lógica de interacción.

Coreografía Modelo de composición de Servicios Web distribuido. Define qué conversaciones se pueden producir entre distintos Servicios Web.

Forja Sitio web dedicado a hospedar proyectos de desarrollo de software. Normalmente, las forjas son gratuitas para proyectos de software libre y/o código abierto, pero también existen forjas para entornos comerciales. Los servicios proporcionados varían mucho entre forja y forja, pero como mínimo suelen tener espacio en algún sistema de control de versiones, un sistema de control de incidencias y un área en la que alojar ficheros para su descarga por los usuarios.

1.5. Otras tecnologías

Para poder realizar este trabajo son necesarios conocimientos en diversos lenguajes y tecnologías.

WS-BPEL [22] es un lenguaje basado en XML estandarizado por OASIS que permite especificar el comportamiento de un proceso de negocio basado en interacciones con WS.

La estructura de un proceso WS-BPEL se divide en cuatro secciones:

1. Definición de relaciones con los socios externos, el cliente que utiliza el proceso de negocio y los WS a los que llama el proceso.
2. Definición de las variables que emplea el proceso.
3. Definición de los distintos tipos de manejadores que puede utilizar el proceso. Pueden definirse manejadores de fallos, que indican las acciones a

realizar en caso de producirse un fallo interno o en un WS al que se llama. También se definen los manejadores de eventos, que especifican las acciones a realizar en caso de que el proceso reciba una petición durante su flujo normal de ejecución.

4. Descripción del comportamiento del proceso de negocio que se logra a través de las actividades que proporciona el lenguaje.

Todos los elementos definidos anteriormente son globales si se declaran dentro del proceso. Pero también se puede declarar de forma local mediante el contenedor `scope`, que permite dividir el proceso de negocio en diferentes ámbitos.

Los principales elementos constructivos son las actividades, que pueden ser básicas y estructuradas. Las actividades básicas son las que realizan una determinada labor (recibir un mensaje, manipular datos, etc.). Las actividades estructuradas pueden contener otras actividades y definen la lógica de negocio.

A las actividades pueden asociarse un conjunto de atributos y un conjunto de contenedores. Estos últimos pueden incluir diferentes elementos que a su vez pueden tener atributos asociados.

Listado 1.1: Ejemplo de una composición WS-BPEL

```
<flow>
  <links>
    <link name="comprobarVuelo-A-reservarVuelo"/>
  </links>
  <invoke name="comprobarVuelo" ...>
  <sources>
    <source linkName="comprobarVuelo-A-reservarVuelo"/>
  </sources>
  </invoke>
  <invoke name="comprobarHotel" ... />
  <invoke name="comprobarAlquilerCoche" ... />
  <invoke name="reservarVuelo" ... />
  <targets>
    <target linkName="comprobarVuelo-A-reservarVuelo" />
  </targest>
  </invoke>
</flow>
```

En el listado 1.1 podemos observar la estructura básica de una composición WS-BPEL. En este ejemplo se ha creado una actividad estructurada con la etiqueta `flow`. Con esta etiqueta se consigue que las actividades se ejecuten en paralelo, por lo tanto, las invocaciones *comprobarVuelo, comprobarHotel, comprobarAlquilerCoche* y *reservarVuelo* se ejecutarían a la vez. Sin embargo no tiene sentido ejecutar la acción *reservarVuelo* antes de *comprobarVuelo*, luego es necesario establecer una sincronización entre algunas actividades, y creamos un enlace entre ambas actividades (líneas 2-4), de forma que *reservarVuelo* no se ejecute hasta que *comprobarVuelo* no haya terminado.

A continuación se muestra una composición que consiste en la aprobación de un préstamo bancario, en la que un cliente solicita un préstamo. Si la cantidad solicitada por el cliente no supera un límite, interviene en la operación un asesor que determinará si el riesgo de dar el préstamo al cliente es alto o bajo. Si es bajo se concede automáticamente, pero si es alto aparece un aprobador que tomará la decisión final. Este aprobador también interviene, y no aparecería el asesor, en el caso de que el cliente pidiera una cantidad que excediese el límite fijado.

Listado 1.2: Ejemplo de la composición del préstamo

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <process
3    name="LoanApprovalProcess"
4    targetNamespace="http://enterprise.netbeans.org/bpel/N6_ServicioPrestamo/
         LoanApprovalProcess"
5    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
6    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7    xmlns:tns="http://enterprise.netbeans.org/bpel/N6_ServicioPrestamo/
         LoanApprovalProcess"
8    xmlns:ns1="http://j2ee.netbeans.org/wsdl/ApprovalService"
9    xmlns:ns2="http://j2ee.netbeans.org/wsdl/AssessorService"
10   xmlns:ns0="http://xml.netbeans.org/schema/Loans"
11   xmlns:ns3="http://j2ee.netbeans.org/wsdl/LoanService">
12   <import namespace="http://j2ee.netbeans.org/wsdl/ApprovalService"
13         location="ApprovalService.wsdl"
14         importType="http://schemas.xmlsoap.org/wsdl/"/>
15   <import namespace="http://j2ee.netbeans.org/wsdl/AssessorService"
16         location="AssessorService.wsdl"
17         importType="http://schemas.xmlsoap.org/wsdl/"/>
```

```
<import namespace="http://j2ee.netbeans.org/wsdl/LoanService"
      location="LoanService.wsdl"
      importType="http://schemas.xmlsoap.org/wsdl/"/>
<partnerLinks>
  <partnerLink name="assessor"
           partnerLinkType="ns2:AssessorService1"
           partnerRole="AssessorServicePortTypeRole"/>
  <partnerLink name="approver"
           partnerLinkType="ns1:ApprovalService1"
           partnerRole="ApprovalServicePortTypeRole"/>
  <partnerLink name="client" partnerLinkType="ns3:LoanService1"
           myRole="LoanServicePortTypeRole"/>
</partnerLinks>
<variables>
  <variable name="processOutput" messageType="ns3:LoanServiceOperationReply"/>
  <variable name="processInput" messageType="ns3:LoanServiceOperationRequest"/>
  <variable name="assessorOutput" messageType="
      ns2:AssessorServiceOperationReply"/>
  <variable name="assessorInput" messageType="
      ns2:AssessorServiceOperationRequest" />
  <variable name="approverOutput" messageType="
      ns1:ApprovalServiceOperationReply"/>
  <variable name="approverInput" messageType="
      ns1:ApprovalServiceOperationRequest"/>
</variables>
<sequence name="Main">
  <receive name="Receive1" createInstance="yes" partnerLink="client"
         operation="grantLoan" portType="ns3:LoanServicePortType"
         variable="processInput"/>
  <assign name="DefaultValues">
    <copy>
      <from>
       <literal><ns0:AssessorRequest><ns0:amount>0.0</ns0:amount></
           ns0:AssessorRequest></literal>
      </from>
      <to part="input" variable="assessorInput"/>
    </copy>
    <copy>
      <from>
       <literal><ns0:ApprovalResponse><ns0:accept>false</ns0:accept></
           ns0:ApprovalResponse></literal>
      </from>
      <to part="output" variable="processOutput"/>
    </copy>
```

```
56      <copy>
57        <from>
58          <literal><ns0:ApprovalRequest><ns0:amount>0.0</ns0:amount></
                ns0:ApprovalRequest></literal>
59        </from>
60        <to part="input" variable="approverInput"/>
61      </copy>
62    </assign>
63    <if name="If1">
64      <condition> ( number(string($processInput.input/ns0:amount)) &lt;= 10000 )
            </condition>
65      <sequence name="SmallAmount">
66        <assign name="copyLoanInfoToAssessorInput">
67          <copy>
68            <from>$processInput.input/ns0:amount</from>
69            <to>$assessorInput.input/ns0:amount</to>
70          </copy>
71        </assign>
72        <invoke name="queryAssessor" partnerLink="assessor"
73              operation="assessLoan" portType="ns2:AssessorServicePortType"
74              inputVariable="assessorInput"
75              outputVariable="assessorOutput"/>
76        <if name="If2">
77          <condition> ( string($assessorOutput.output/ns0:risk) = 'high' ) </
                condition>
78          <sequence name="SmallAmountHighRisk">
79            <assign name="copyLoanInfoToApproverInput">
80              <copy>
81                <from>$processInput.input/ns0:amount</from>
82                <to>$approverInput.input/ns0:amount</to>
83              </copy>
84            </assign>
85            <invoke name="queryApprover" partnerLink="approver"
86                  operation="approveLoan"
87                  portType="ns1:ApprovalServicePortType"
88                  inputVariable="approverInput"
89                  outputVariable="approverOutput"/>
90            <assign name="copyApproval">
91              <copy>
92                <from>$approverOutput.output/ns0:accept</from>
93                <to>$processOutput.output/ns0:accept</to>
94              </copy>
95            </assign>
96          </sequence>
```

```
97        <else>
98          <sequence name="SmallAmountLowRisk">
99            <assign name="approveLoan">
100             <copy>
101               <from>true()</from>
102               <to>$processOutput.output/ns0:accept</to>
103             </copy>
104           </assign>
105         </sequence>
106       </else>
107     </if>
108   </sequence>
109   <else>
110     <sequence name="LargeAmount">
111       <assign name="copyLoanInfoToApproverInput2">
112         <copy>
113           <from>$processInput.input/ns0:amount</from>
114           <to>$approverInput.input/ns0:amount</to>
115         </copy>
116       </assign>
117       <invoke name="queryApprover2"
118             partnerLink="approver"
119             operation="approveLoan"
120             portType="ns1:ApprovalServicePortType"
121             inputVariable="approverInput"
122             outputVariable="approverOutput"/>
123       <assign name="copyApproval2">
124         <copy>
125           <from>$approverOutput.output/ns0:accept</from>
126           <to>$processOutput.output/ns0:accept</to>
127         </copy>
128       </assign>
129     </sequence>
130   </else>
131  </if>
132  <reply name="Reply1" partnerLink="client" operation="grantLoan"
133       portType="ns3:LoanServicePortType" variable="processOutput"/>
134 </sequence>
135 </process>
```

WS-BPEL es un lenguaje de orquestación, no uno basado en coreografías. La diferencia entre ambos es que en los modelos basados en coreografías, se representan todos los actores del sistema, con sus interacciones, dando una perspectiva glo-

bal del sistema. Sin embargo, en los modelos de orquestación, se centra en un participante en particular.

Las composiciones WS-BPEL pueden ser de dos tipos: síncronas y asíncronas. La diferencia entre ambas es que en las composiciones síncronas el cliente envía una petición y espera a que se envíe la respuesta sobre la misma conexión, y las composiciones asíncronas la propia composición le invoca más tarde.

BPELUnit [18] es un marco de pruebas unitarias para hacer pruebas de composiciones WS-BPEL. Está basado en XML (fichero con extensión "bpts"), luego es portable y se puede ejecutar en cualquier motor WS-BPEL. Puede describir los casos de prueba a ejecutar y la posibilidad de sustituir servicios externos con otros servicios (*mockups*) que los simulen desarrollando el comportamiento indicado en la especificación proporcionada por el usuario. Además, BPELUnit ofrece la posibilidad de añadir envío síncronos y asíncronos.

Un fichero *bpts* define qué proceso se va a probar y cómo. Al estar basado en XML, puede validarse e incluye un elemento raíz donde se especifican los espacios de nombres.

Como podemos ver en la figura 1.2, el documento está estructurado en dos partes fundamentales:

1. ***Sección de despliegue***: contiene la información para el despliegue y la especificación de todos los *partners* (servicios a los que se conecta). Contiene los siguientes elementos:

   name Un nombre identificativo para el proceso.

   baseURL La URL sobre la que se desplegará.

   deployment La información sobre el despliegue del proceso.

   - **put** El proceso a desplegar. Se especifica su nombre, el motor que lo ejecutará, el fichero WSDL que describe el servicio y el fichero `BPR` que contiene la composición.
   - **partner** Los servicios externos a los que se conectará, junto a su descripción WSDL. Estos servicios externos pueden sustituirse por *moc-*

kups cuando se especifique el caso de prueba. El atributo `name` debe identificar unívocamente al *partner*, ya que será el nombre que se usará después para sustituirlo por uno simulado.

2. ***Sección de casos de prueba***: contiene una serie de casos de prueba. Se especifican como un elemento raíz *testCases* en el que se anidan elementos *testCase* para cada caso de prueba.

   testCase El elemento para definir un caso de prueba. El atributo `name` es un identificador único del caso de prueba. Un caso de prueba puede basarse en uno anterior, definido con el atributo `basedOn`. Si el atributo `abstract` tiene el valor "true" entonces el caso de prueba no puede ser ejecutado. Por último, `vary` especifica si se debe ejecutar más de una vez el mismo caso de prueba cambiando tiempos de respuesta.

   Cada caso de prueba contiene un elemento `clientTrack` y cero o varios `partnerTrack`:

   clientTrack Define la petición que se enviará desde el cliente simulado.

   partnerTrack Se utiliza para definir un *mockup*. La estructura es prácticamente idéntica al `clientTrack`. El atributo `name` define qué servicio externo está siendo reemplazado. Cada caso de prueba debe contener un único `partnerTrack` por cada *partner*.

Tanto los `clientTrack` como los `partnerTrack` pueden contener una secuencia de actividades soportadas por BPELUnit:

sendOnly Esta actividad envía un mensaje simple. En el este bloque los datos incluidos dentro de *data* se enviarán al proceso. Estos datos seguirán la estructura definida por el propio proceso. El atributo `fault` a "false" indicará que no debe simularse un fallo, y a "true" que sí.

receiveOnly Esta actividad espera por un mensaje simple y lo verifica. Este bloque se evaluará cuando el proceso responda al cliente. Con *condition* pueden comprobarse si ciertas condiciones esperadas en el mensaje de respuesta se cumplen, si esta condición no se evalúa como verdadera para la actividad no pasará la prueba.

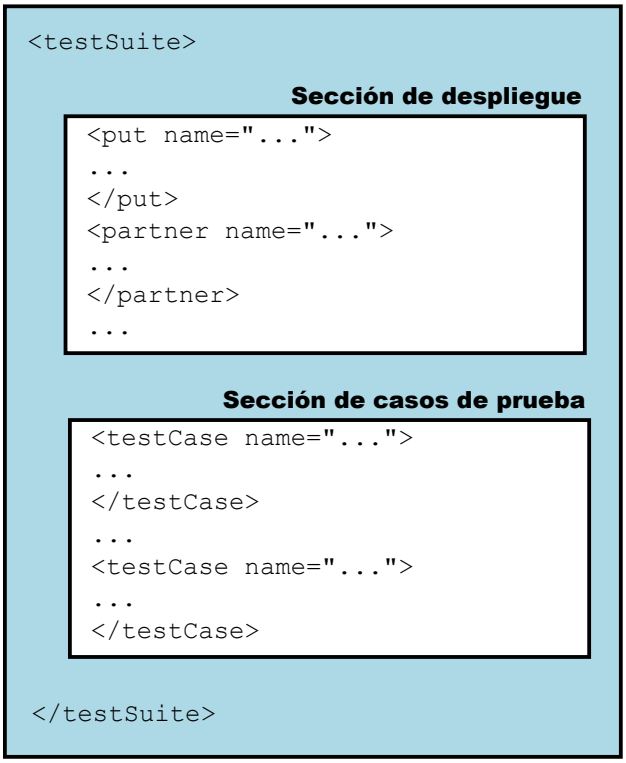

Figura 1.2: Estructura de un fichero *bpts*

sendReceive En el caso de `clientTrack` el cliente del proceso mandará una petición síncrona. Los atributos especifican a qué servicio y puerto se enviará la petición, y qué operación se usará.

receiveSend Espera un mensaje simple, lo verifica, y envía una respuesta de vuelta sincronizada.

receiveSendAsynchronous Se reciben primero los datos, y se responde de forma asíncrona, por tanto, el proceso cliente no espera la respuesta.

sendReceiveAsynchronous Envía un mensaje simple, espera una respuesta asíncrona, y verifica la respuesta.

Listado 1.3: Ejemplo de un fichero *bpts*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tes:testSuite
3    xmlns:esq="http://xml.netbeans.org/schema/Loans"
```

```
4    xmlns:ap="http://j2ee.netbeans.org/wsdl/ApprovalService"
5    xmlns:as="http://j2ee.netbeans.org/wsdl/AssessorService"
6    xmlns:sp="http://j2ee.netbeans.org/wsdl/LoanService"
7    xmlns:pr="http://enterprise.netbeans.org/bpel/N6_ServicioPrestamo/
         LoanApprovalProcess"
8    xmlns:tes="http://www.bpelunit.org/schema/testSuite">
9
10   <tes:name>LoanServiceTest</tes:name>
11   <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
12
13   <tes:deployment>
14    <tes:put name="LoanApprovalProcess" type="activebpel">
15     <tes:wsdl>LoanService.wsdl</tes:wsdl>
16     <tes:property name="BPRFile">LoanApprovalDoc.bpr</tes:property>
17    </tes:put>
18    <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
19    <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
20   </tes:deployment>
21
22   <tes:testCases>
23    <tes:testCase name="SmallAmountHighRiskRejected"
24              basedOn="" abstract="false" vary="false">
25
26     <tes:clientTrack>
27      <tes:sendReceive
28         service="sp:LoanServiceService"
29         port="LoanServicePort"
30         operation="grantLoan">
31       <tes:send fault="false">
32        <tes:data>
33         <esq:ApprovalRequest>
34          <esq:amount>1500</esq:amount>
35         </esq:ApprovalRequest>
36        </tes:data>
37       </tes:send>
38       <tes:receive fault="false">
39        <tes:condition>
40         <tes:expression>esq:ApprovalResponse/esq:accept</tes:expression>
41         <tes:value>'false'</tes:value>
42        </tes:condition>
43       </tes:receive>
44      </tes:sendReceive>
45     </tes:clientTrack>
46
```

```
47      <tes:partnerTrack name="approver">
48        <tes:receiveSend
49           service="ap:ApprovalServiceService"
50           port="ApprovalServicePort"
51           operation="approveLoan">
52          <tes:send fault="false">
53            <tes:data>
54              <esq:ApprovalResponse>
55                <esq:accept>false</esq:accept>
56              </esq:ApprovalResponse>
57            </tes:data>
58          </tes:send>
59          <tes:receive fault="false"/>
60        </tes:receiveSend>
61      </tes:partnerTrack>
62
63      <tes:partnerTrack name="assessor">
64        <tes:receiveSend
65           service="as:AssessorServiceService"
66           port="AssessorServicePort"
67           operation="assessLoan">
68          <tes:receive fault="false"/>
69          <tes:send fault="false">
70            <tes:data>
71              <esq:AssessorResponse>
72                <esq:risk>high</esq:risk>
73              </esq:AssessorResponse>
74            </tes:data>
75          </tes:send>
76        </tes:receiveSend>
77      </tes:partnerTrack>
78
79    </tes:testCase>
80  </tes:testCases>
81 </tes:testSuite>
```

En el listado 1.3 se puede observar la estructura general de un fichero *bpts* mediante un fichero de pruebas de la típica composición de WS-BPEL consitente en la aprobación de un préstamo (*Loan Approval*). Este ejemplo contiene exactamente una prueba, ya que se ha definido un solo `<testCase>`. En el caso de que se quisieran realizar más pruebas con distintos valores, o que intervengan distintos servicios, habría que definir dentro del mismo `<testCases>` varios

`<testCase>`.

En esta única prueba intervienen tres entidades: un cliente, un asesor y un aprobador. El cliente solicita un préstamo; dependiendo de la cantidad solicitada, el proceso WS-BPEL invoca al asesor (WS *assessor*) si la cantidad es menor que un cierto valor o al aprobador (WS *approver*) en otro caso. La respuesta o salida del asesor se corresponde con el nivel de riesgo del cliente; si el riesgo es bajo, se concede el préstamo, si es alto, se invoca al aprobador, cuya respuesta o salida se corresponderá con la decisión final de aceptación del préstamo.

Se pueden observar las dos secciones definidas anteriormente, la sección de despliegue (líneas 10-20) y la sección de casos de prueba (líneas 22-82).

En este caso lo que se quiere probar es que un cliente solicita un préstamo de 1.500 €; el asesor decide que el riesgo del cliente es alto; y finalmente el aprobador responde con la no aceptación del préstamo (“false”).

Se puede comprobar en el *sendReceive* (línea 27) del cliente, en el apartado *value* del *receive* (línea 41) se espera exactamente que se deniegue el préstamo. Si la composición es correcta, debería pasar la prueba con éxito.

WSDL [29] es un lenguaje basado en XML utilizado para describir la interfaz de WS: qué pueden hacer, dónde se encuentran y qué tipo de datos esperan y en qué formato. Una composición WS-BPEL normalmente reúne varios WS en uno de nivel superior.

WSDL consta de un conjunto muy amplio de reglas y normas. En la práctica, no se suelen implementar todas: hacerlo sería muy complejo y costoso. La mayoría de las implementaciones actuales se centran en el subconjunto definido por la especificación WS-I Basic Profile 1.1 [32]. Este subconjunto permite conseguir una mayor interoperabilidad entre todas las implementaciones existentes.

A continuación, definimos la estructura general de WSDL 1.1:

Types Contiene las definiciones de los tipos de datos, normalmente utilizando un sistema de tipo XSD.

Message Define el formato de los mensajes que se intercambian entre el servicio Web y el cliente del servicio Web. Está compuesto de una o más partes lógicas. Cada parte puede referirse a un elemento XSD y/o a tipos XSD simples o complejos.

Operation Consiste en un parámetro o parámetros de entrada, un valor de salida y un elemento de fallo opcional.

Input Define la entrada de la operación.

Output Define el valor de salida de la operación.

Fault Mensaje de excepción que se devuelve desde el servicio Web al que ha llamado, en caso de que se capture un error.

Una operación WSDL 1.1 soporta 4 tipos de patrones de intercambio de mensajes:

One-way El *endpoint* recibe un mensaje.

Request-response El *endpoint* recibe un mensaje y devuelve un mensaje correlacionado al que ha llamado.

Solicit-response El *endpoint* envía un mensaje y recibe una respuesta correlacionada.

Notification El *endpoint* envía un mensaje.

Binding Contiene los detalles de cómo los elementos abstractos de un *portType* se limitan a un concreto conjunto de elementos. El protocolo de comunicación más utilizado es HTTP y el de representación de mensajes entre las partes es SOAP.

Service Contenedor que agrupa los elementos *port*.

Port Especifica la dirección, en concreto la URL, para un *binding*.

SOAP [30] (Simple Object Access Protocol) es un protocolo estándar que define como dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML. Este protocolo deriva de un protocolo creado por David Winer en 1998, llamado XML-RPC. SOAP fue creado por Microsoft, IMB y otros y está ac-

tualmente bajo el auspicio de la W3C. Es uno de los protocolos utilizados en los Servicios Web.

XML Schema [31] es un lenguaje utilizado para describir la estructura y las restricciones de los contenidos de los documentos XML. De esta forma podemos controlar la estructura y los tipos de datos de una manera muy amplia. Podemos ver un ejemplo en el listado 1.4 en el que tenemos un libro definido por cuatro elementos: título, autores, edición y número de páginas, todos ellos de tipo cadena.

Listado 1.4: Ejemplo del XML Schema de un libro

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="libro">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Titulo" type="xs:string"/>
   <xs:element name="Autores" type="xs:string" maxOccurs="10"/>
   <xs:element name="Editorial" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="precio" type="xs:double"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Y un ejemplo de fichero XML asignado al XML Schema sería el siguiente:

Listado 1.5: Ejemplo de fichero XML asignado al XML Schema

```
<?xml version="1.0"?>
<libro xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="libro.xsd" precio="20">
  <Titulo>Fundamentos de XML Schema</Ttulo>
  <Autores>Allen Wyke</Autores>
  <Autores>Andrew Watt</Autores>
  <Editorial>Wiley</Editorial>
</libro>
```

Apache Velocity [10] es un motor de plantillas basado en Java. Le permite a los diseñadores de páginas hacer referencia a métodos definidos dentro del código Java. Los diseñadores Web pueden trabajar en paralelo con los programadores Java

para desarrollar sitios de acuerdo al modelo de Modelo-Vista-Controlador (MVC), permitiendo que los diseñadores se concentren únicamente en crear un sitio bien diseñado y que los programadores se encarguen solamente de escribir código de primera calidad. Velocity separa el código Java de las páginas Web, haciendo el sitio más mantenible a largo plazo y presentando una alternativa viable a Java Server Pages (JSP) o PHP.

BPELUnit utiliza Velocity de dos formas: como lenguaje de plantillas propiamente dicho para generar los mensajes a enviar a la composición WS-BPEL, y como formato de entrada para recibir los valores de las variables a usar en las plantillas de los mensajes.

La primera forma la utilizaremos dentro de los *bpts*. Recuérdese que para crear varias pruebas había que definir dentro de la sección `<testCases>` varios `<testCase>` y para cada prueba había que duplicar buena parte del código, cambiando sólo el valor de algunas variables. Pues bien, utilizando Velocity podemos describir una única vez la estructura que tendrán las pruebas, y para cada prueba, determinadas variables tomarán una serie de valores previamente indicados.

Veamos para el ejemplo del *bpts* que probaba la composición del aprobador, qué aspecto tendría si utilizáramos Velocity:

Listado 1.6: Ejemplo de un fichero *bpts* basado en plantillas Velocity

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tes:testSuite
3     xmlns:esq="http://xml.netbeans.org/schema/Loans"
4     xmlns:ap="http://j2ee.netbeans.org/wsdl/ApprovalService"
5     xmlns:as="http://j2ee.netbeans.org/wsdl/AssessorService"
6     xmlns:sp="http://j2ee.netbeans.org/wsdl/LoanService"
7     xmlns:pr="http://enterprise.netbeans.org/bpel/N6_ServicioPrestamo/
          LoanApprovalProcess"
8     xmlns:tes="http://www.bpelunit.org/schema/testSuite">
9
10   <tes:name>LoanServiceTest</tes:name>
11   <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
12
13   <tes:deployment>
14     <tes:put name="LoanApprovalProcess" type="activebpel">
15       <tes:wsdl>LoanService.wsdl</tes:wsdl>
```

```
16      <tes:property name="BPRFile">LoanApprovalDoc.bpr</tes:property>
17    </tes:put>
18    <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
19    <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
20  </tes:deployment>
21
22  <tes:testCases>
23    <tes:testCase name="MainTemplate" basedOn="" abstract="false" vary="false">
24      <tes:setUp>
25        <tes:dataSource type="velocity" src="data.vm">
26          <tes:property name="␣iteratedVars">
27            cantidad aprobador riesgo aceptado
28          </tes:property>
29        </tes:dataSource>
30      </tes:setUp>
31
32      <tes:clientTrack>
33        <tes:sendReceive
34           service="sp:LoanServiceService"
35           port="LoanServicePort"
36           operation="grantLoan">
37          <tes:send fault="false">
38            <tes:template>
39              <esq:ApprovalRequest>
40                <esq:amount>$cantidad</esq:amount>
41              </esq:ApprovalRequest>
42            </tes:template>
43          </tes:send>
44          <tes:receive fault="false">
45            <tes:condition>
46              <tes:expression>esq:ApprovalResponse/esq:accept</tes:expression>
47              <tes:value>$aceptado</tes:value>
48            </tes:condition>
49          </tes:receive>
50        </tes:sendReceive>
51      </tes:clientTrack>
52
53      <tes:partnerTrack name="approver">
54        <tes:receiveSend
55           service="ap:ApprovalServiceService"
56           port="ApprovalServicePort"
57           operation="approveLoan"
58           assume="$aprobador␣!=␣'silent'">
59          <tes:send fault="false">
```

```
60          <tes:template>
61            <esq:ApprovalResponse>
62              <esq:accept>$aprobador</esq:accept>
63            </esq:ApprovalResponse>
64          </tes:template>
65        </tes:send>
66
67        <tes:receive fault="false">
68          <tes:condition>
69            <tes:expression>//esq:amount</tes:expression>
70            <tes:value>$cantidad</tes:value>
71          </tes:condition>
72        </tes:receive>
73      </tes:receiveSend>
74    </tes:partnerTrack>
75
76    <tes:partnerTrack name="assessor" assume="$riesgo␣!=␣'silent'">
77      <tes:receiveSend
78         service="as:AssessorServiceService"
79         port="AssessorServicePort"
80         operation="assessLoan">
81
82        <tes:receive fault="false">
83          <tes:condition>
84            <tes:expression>//esq:amount</tes:expression>
85            <tes:value>$cantidad</tes:value>
86          </tes:condition>
87        </tes:receive>
88
89        <tes:send fault="false">
90          <tes:template>
91            <esq:AssessorResponse>
92              <esq:risk>$riesgo</esq:risk>
93            </esq:AssessorResponse>
94          </tes:template>
95        </tes:send>
96      </tes:receiveSend>
97    </tes:partnerTrack>
98
99   </tes:testCase>
100  </tes:testCases>
101 </tes:testSuite>
```

La primera diferencia que se observa es que hay una nueva sección llamada

`<dataSource>` donde se indica el formato que se va a utilizar, el fichero donde están almacenados los valores de las variables, y el nombre de las variables.

También se aprecia que en las secciones `<send>` hay una nueva etiqueta llamada `<template>` donde se accede al valor de la variable precediendo al nombre de la variable el símbolo $.

Usaremos también Velocity para definir en un fichero *vm* el catálogo de pruebas.

Cada fila del fichero Velocity comienza de la forma `#set`, seguido del nombre de la variable correspondiente y a continuación los distintos valores que toma esa variable para cada uno de los casos de prueba. Cada caso de prueba se corresponde con una columna en particular.

Continuando con la composición de la aprobación de un préstamo, podríamos definir el fichero *vm* tal y como se muestra en el siguiente listado.

Listado 1.7: Ejemplo de un fichero *vm*

```
1 #set($cantidad = [150000, 1500, 3000, 6000, 8000])
2 #set($aprobador = ["true", "false", "silent", "false", "true"])
3 #set($riesgo = ["silent", "high", "low", "high", "low"])
4 #set($aceptado = ["true", "false", "true", "false", "true"])
```

spec Como se dijo anteriormente, los ficheros *vm* pueden crearse a mano, sabiendo de antemano las pruebas que se desean realizar, o bien pueden crearse de forma aleatoria a partir de otro fichero con extensión “spec” [24]. Recordemos que estos ficheros *spec* almacenan tanto los tipos como las variables de los servicios que intervienen en la composición. Siguiendo con el ejemplo de la composición del aprobador, un *spec* relacionado con la misma sería el listado 1.8.

Listado 1.8: Ejemplo de un fichero *spec*

```
1 typedef int (min=0, max = 200000) Cantidad;
2 typedef string (values={"true", "false", "silent"}) Aprobador;
3 typedef string (values={"low", "high", "silent"}) Asesor;
4 typedef string (values={"true", "false"}) Aceptado;
5
6 Cantidad cantidad;
7 Aprobador aprobador;
8 Asesor riesgo;
```

```
9 Aceptado aceptado;
```

Estos ficheros se dividen en dos partes: **Typedefs** donde se definen los tipos con sus restricciones, si procede, y **Variables** donde se definen las variables en base a los tipos previamente declarados.

Por ejemplo en la línea 1 del listado 1.8 tenemos el `typedef` llamado `Cantidad` que es de tipo entero con las restricciones valor mínimo y máximo `min=0, max=200000` respectivamente. En la línea 6 se crea una variable de este nuevo tipo creado llamada `cantidad`. Se pueden crear más variables de un mismo tipo de las siguientes maneras:

```
Cantidad cantidad1, cantidad2;
Cantidad cantidad1;
Cantidad cantidad2;
```

El conjunto de datos capaz de generar de forma aleatoria es el siguiente:

- Enteros.
- Números con coma flotante.
- Cadena de caracteres.
- Fechas.
- Horas.
- Fechas y horas.
- Duraciones.
- Listas.
- N-Tuplas.

Con las siguientes restricciones:

- Valor máximo.
- Valor mínimo.
- Número total de dígitos.

- Número total de decimales.
- Cumplir cierta expresión regular.
- Valor válido dada una lista de valores.

Si al *spec* de la figura 1.8 posteriormente ejecutáramos la herramienta TestGenerator, generaría un *vm* con datos aleatorios (listado 1.9).

Listado 1.9: *vm* resultante tras ejecutar TestGenerator

```
1 #set($cantidad = [181085, 108959, 42020, 58951, 183065])
2 #set($aprobador = ["silent", "true", "silent", "true", "false"])
3 #set($riesgo = ["silent", "silent", "silent", "silent", "high"])
4 #set($aceptado = ["false", "true", "true", "true", "false"])
```

1.6. Estructura del trabajo

En la presente memoria se detalla, en cada capítulo, las distintas fases de la creación del proyecto, así como las tecnologías utilizadas y las pruebas realizadas al mismo.

En el Capítulo 2 se explica la manera en la que se ha organizado el tiempo para desarrollar este proyecto: metodología usada, etapas y el diagrama de Gantt.

En el Capítulo 3 se da una introducción a los conceptos básicos sobre pruebas de software. Aparte de definir estos conceptos sobre pruebas, se habla sobre las estrategias de aplicación de las pruebas, de las técnicas de prueba del software y de las pruebas de mutaciones, lo cual es fundamental para comprender el propósito de este proyecto.

En el Capítulo 4 comenzamos a analizar el proyecto: requisitos funcionales, requisitos de implementación, atributos del sistema, casos de uso, etc. Se indaga en el análisis de una composición WS-BPEL. En este análisis se detallan los conceptos de las composiciones WS-BPEL necesarios para crear plantillas que prueben dichas composiciones.

En el Capítulo 5 pasamos al diseño del proyecto. En este capítulo hablamos de la arquitectura que se ha elegido para desarrollar la herramienta, se muestra el diagrama de clases del sistema y se detallan otros componentes necesarios para implementar el código, tales como TestGenerator, ServiceAnalyzer, XMLBeans, el framework de BPEL-Unit, etc.

En el Capítulo 6 hablaremos de los criterios de implementación seguidos, tecnologías y herramientas útiles para el desarrollo de cualquier software y el plan de pruebas que se ha seguido en el proyecto.

Por último, en los Anexos A y B podemos ver unos manuales con los que poder hacer uso de la herramienta y cómo poder acceder al código fuente para desarrolladores.

Capítulo 2

Calendario

A continuación se detalla la manera en la que se ha organizado temporalmente el proyecto, las tareas realizadas y la planificación de las mismas.

La idea del proyecto surgió en diciembre de 2011. En el curso académico 2011/2012 el alumno fue seleccionado como alumno colaborador del profesor Juan José Domínguez Jiménez, y entró a formar parte del grupo de investigación UCASE. Hasta la terminación del curso académico el alumno comenzó a familiarizarse con las tecnologías con las que trabajaba el grupo, los proyectos creados y asistía a los seminarios semanales donde alumnos y profesores exponían sus trabajos e investigaciones.

Desde el comienzo del curso 2012/2013, se ha centrado más el tiempo en la preparación del proyecto, sin dejar de lado la colaboración con el grupo en la prueba de composiciones WS-BPEL. Se comenzó a estudiar las composiciones del repositorio público que tiene el grupo de investigación en Redmine[1], desde las composiciones más sencillas a las más complejas, o las que incluían mas detalles a tener en cuenta a la hora de analizar las composiciones WS-BPEL.

2.1. Metodología

Se ha usado una metodología iterativa basada en prototipos siguiendo el modelo de ciclo de vida incremental. Se crearon unos requisitos iniciales que posteriormente

[1] `https://neptuno.uca.es/redmine/`

se fueron incrementando hasta obtener el producto final.

Esta metodología se adapta a las necesidades del proyecto ya que aunque el objetivo final del proyecto estaba bien definido, no se sabía de antemano todas las características necesarias para la construcción de la herramienta. Además se utilizan funcionalidades de otros proyectos (TestGenerator, ServiceAnalyzer, etc), los cuales continúan hoy en día en desarrollo, así que si se modificaba alguno de estos proyectos habría que modificar también BPTSGenerator.

2.2. Etapas

2.2.1. Elicitación de requisitos

Los requisitos se analizaron a través de reuniones con los profesores tutores del proyecto. Sin embargo se fueron refinando a lo largo de todo el proyecto.

2.2.2. Estudio de lenguajes y tecnologías

Una vez obtenidos los requisitos del proyecto se pasó a estudiar las tecnologías, herramientas y lenguajes que se han utilizado para la realización de este proyecto.

Fue necesario un gran esfuerzo por parte del alumno, ya que la mayoría de las tecnologías y lenguajes utilizados no las había usado con anterioridad, alguna de las cuales se encuentran incluso en fase experimental. Además del estudio de otras herramientas que iba a necesitar comprender por necesitar alguna de sus funcionalidades.

2.2.3. Análisis de la composición LoanApprovalDoc

Esta composición fue la primera que se estudiaba en detalle, tanto el comportamiento de la composición con sus actividades, como los ficheros WSDL y el *bpts* que ya existía creado a mano por los miembros del grupo UCASE. Así que fue la que más trabajo costó analizar.

Fue necesario modificar un pequeño problema en el proyecto ServiceAnalyzer que había con unos saltos de línea indeseados en la plantilla Velocity de catálogo que devolvía la herramienta al analizar un fichero WSDL.

2.2.4. Análisis de la composición LoanApprovalRPC

No hizo falta realizar ningún cambio, ya que la única diferencia entre LoanApprovalDoc y LoanApprovalRPC está en los ficheros WSDL, y del análisis de éstos se encargaba ServiceAnalyzer.

2.2.5. Análisis de la composición MarketPlace

Para analizar esta composición fue necesario familiarizarse con el concepto de conjunto de correlación (que se explicará más adelante). Este concepto es sencillo de entender pero fue difícil de implementar. Hubo que modificar algunas clases de un proyecto desarrollado por los miembros del grupo UCASE llamado bpel-packager, en el que se desarrollan utilidades para manejar los distintos tipos de ficheros que se usan en varios proyectos del grupo. Estas modificaciones fueron necesarias para extraer una información de los ficheros WSDL, fundamental para tratar los conjuntos de correlación, y que hasta ahora no le había hecho falta a nadie.

En esta composición también se tubo en cuenta que puede haber varias actividades `<receive>` y hubo que estudiar cuál es la que actúa como cliente y cuáles no.

2.2.6. Análisis de la composición MarketPlaceFlow

En esta composición apareció otra actividad a tener en cuenta distinta de las ya estudiadas en las composiciones anteriores llamada `<onMessage>`. Se comprobó que si crea una instancia de la misma había que tenerla en cuenta y su comportamiento es similar al de las actividades `<receive>`.

2.2.7. Análisis de la composición SquaresSum

No hizo falta ninguna modificación cuando se estudió esta composición. Sin embargo, se detectó un problema en TestGenerator al validar un *spec*. Esta herramienta tiene una función que valida el fichero *spec* y daba problemas con el valor máximo permitido por un elemento de tipo entero sin signo.

2.2.8. Análisis de la composición ShippingSync

No fue necesario modificar la herramienta, aunque se detectó un error en ServiceAnalyzer, ya que cuando se restringían los valores de un tipo numérico entre un valor máximo y un mínimo, si el tipo era *float* ServiceAnalyzer lo interpretaba como *int* y los casos de prueba generados eran erróneos.

2.2.9. Composición nueva de LoanApproval con dos asesores

Se identificó la situación de que un mismo servicio puede llamarse varias veces en la composición, con lo que habría que probar el mismo servicio con variables distintas, y este aspecto no se detectaba con la versión de la herramienta. Se consultó con un tutor del proyecto y se decidió dejarlo como trabajo futuro ya que realizar un análisis más exhaustivo a la composición significaba invertir mucho más tiempo ya que habría que recorrer todos los posibles caminos de una composición, y para composiciones complejas este trabajo es muy complicado. Así que se decidió restringir esta situación a declarar dos variables distintas de un mismo servicio en la composición.

Ya que no había una composición en la que se diera esta situación, el alumno modificó LoanApproval para que aceptara dos asesores identificados con variables distintas.

2.2.10. Pruebas

El proyecto está implementado en Java y se utilizó el framework JUnit [15] para poder realizar pruebas al mismo y conseguir un software de mejor calidad.

2.2.11. Paquete de instalación y línea de órdenes

En esta etapa se hicieron los ajustes necesarios para que BPTSGenerator pueda ser fácilmente instalable en el equipo de un futuro usuario y pueda utilizarlo desde la línea de órdenes.

2.2.12. Documentación

La recopilación de la información para escribir esta memoria se ha ido recogiendo a lo largo de la creación del proyecto. Sin embargo, la escritura de la misma se ha

realizado en las últimas semanas.

2.3. Diagrama de Gantt y fases del proyecto

Se ha elaborado un diagrama de Gantt (ver figuras 2.1 y 2.2) para poder visualizar con mayor facilidad la distribución de las tareas.

Se ha empleado la herramienta Gantt Project para dibujar el diagrama. Esta herramienta es de código abierto, está hecha en Java y disponible en `http://ganttproject.sourceforge.net`.

En la tabla 2.1 se muestran las distintas fases con sus respectivas fechas de inicio y final.

Fase	Día inicio	Día fin
Elicitación de requisitos	21/03/12	26/04/12
Estudio de lenguajes y tecnologías	2/03/12	20/12/12
Análisis de LoanApprovalDoc	10/10/12	8/03/13
Análisis de LoanApprovalRPC	1/03/13	10/3/13
Análisis de MarketPlace	8/3/13	18/5/13
Análisis de MarketPlaceFlow	25/3/13	28/5/13
Análisis de SquaresSum	28/5/13	31/5/13
Análisis de ShippingSync	30/5/13	2/6/13
Análisis de LoanApproval dos asesores	3/6/13	4/6/13
Pruebas	1/3/13	4/6/13
Paquete de instalación	4/6/13	4/6/13
Documentación	1/5/13	14/6/13

Tabla 2.1: Fases del proyecto

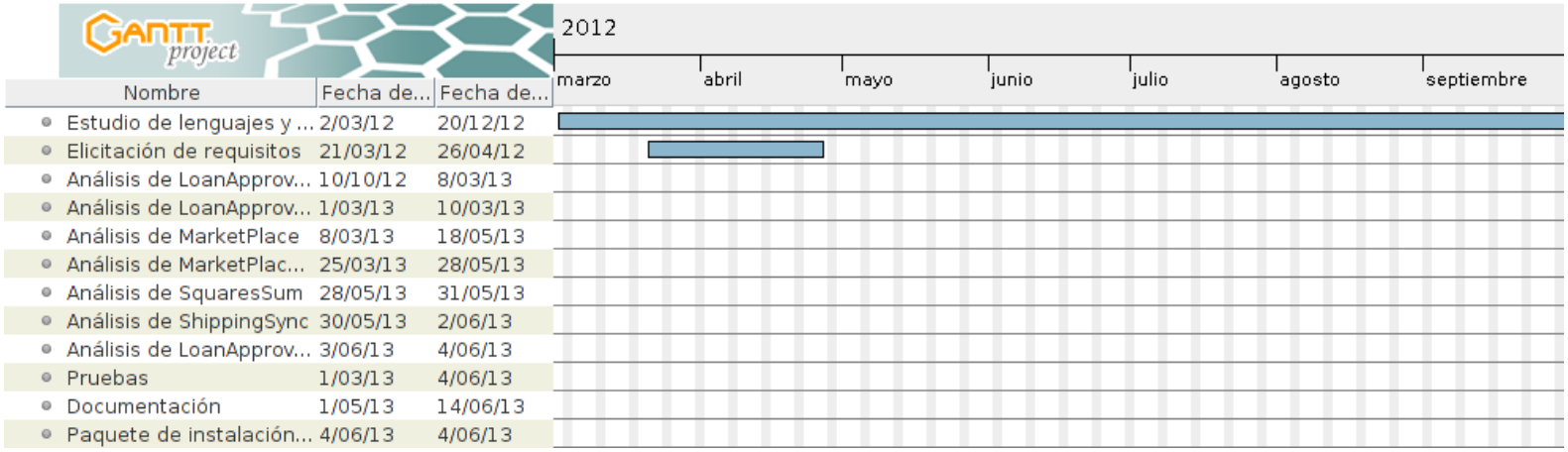

Figura 2.1: Diagrama de Gantt, primera parte

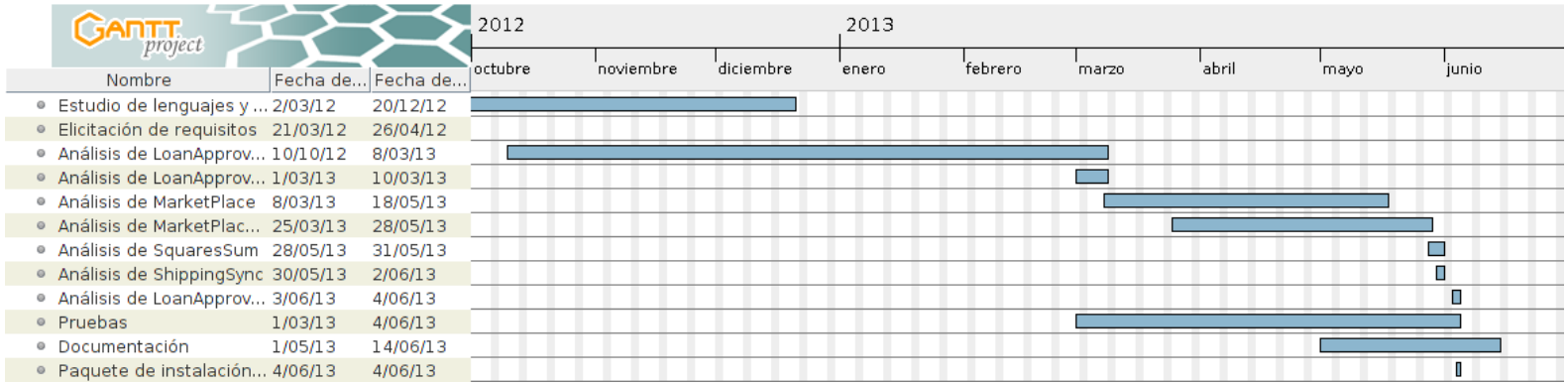

Figura 2.2: Diagrama de Gantt, segunda parte

Capítulo 3

Pruebas de software

La prueba es un conjunto de actividades que se planean con anticipación y se realizan de manera sistemática. Por tanto, se deben definir un conjunto de pasos en que se puedan incluir técnicas y métodos específicos del diseño de casos de prueba para las pruebas del software.

3.1. Introducción

En el diccionario de IEEE [28] podemos encontrar las siguientes definiciones:

Prueba "Una actividad en la cual un sistema o alguno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran y se realiza una evaluación de algún aspecto".

Caso de prueba "Un conjunto de entradas, condiciones de ejecución, y resultados esperados desarrollados para un objetivo particular como, por ejemplo, probar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito".

Por lo tanto el objetivo de las pruebas es la detección de defectos en el software, y descubrir un defecto debería considerarse el éxito de la prueba.

Por otro lado, Myers [21] (capítulo 2, página 11) define las pruebas de software como:

> "Hacer pruebas es el proceso de ejecutar un programa con la intención de encontrar errores."

Esta definición es simple pero precisa, y determina cuál debe ser la actitud de una persona que se dedique a probar software. No es lo mismo probar que un software funciona correctamente (no contiene errores) que demostrar que no funciona (tiene errores).

La definición de Dijkstra [4] (página 6) es más determinante:

> "Las pruebas del software pueden probar la presencia de errores pero no la ausencia de ellos."

Sin embargo, la definición que más se ajusta a la realidad del programador es la de que las pruebas de software consisten en verificar el comportamiento del software a partir de una serie de casos de prueba [25]. Es decir, ejecutar al sistema que se desea probar una serie de casos de prueba para verificar si funciona como debería.

3.2. El proceso de prueba

En la figura 3.1 se muestra el proceso completo de pruebas. El proceso comienza con la generación de un plan de pruebas en base a la documentación del proyecto y a la documentación del software a probar. A partir de dicho plan, se diseñan los casos de prueba, se ejecutan y los resultados obtenidos se comparan con los resultados esperados. Una vez evaluados los resultados de las pruebas, pueden realizarse dos actividades:

- Depurar los defectos.
- Analizar los errores.

La depuración puede corregir o no los defectos. Si no consigue localizarlos, puede ser necesario realizar pruebas adicionales para obtener más información. Si se corrige un defecto, se debe volver a probar el software para comprobar que el problema está resuelto.

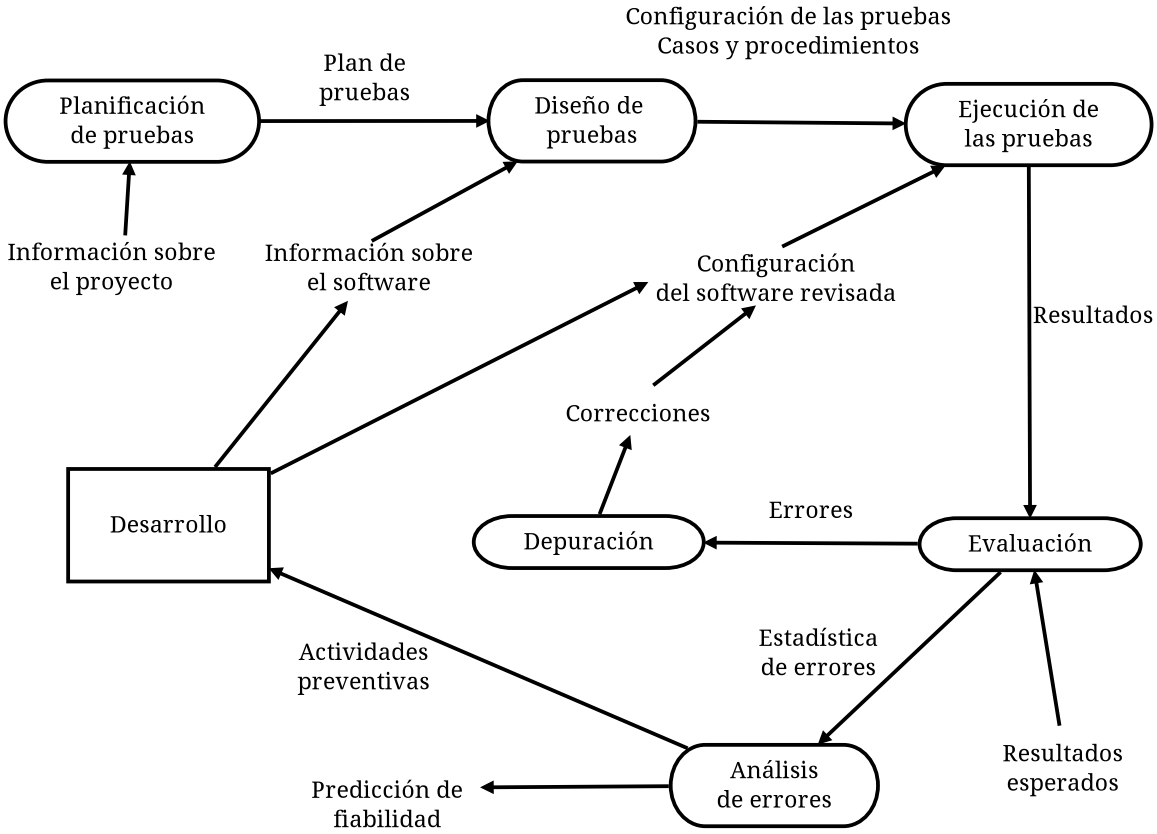

Figura 3.1: Proceso de las pruebas

Por otra parte, el análisis de errores puede servir para realizar predicciones de la fiabilidad del software y para detectar las causas más habituales de error y mejorar los procesos de desarrollo.

3.3. Estrategias de aplicación de las pruebas

La estrategia de aplicación y planificación de las pruebas tiene como finalidad crear distintos niveles de prueba con diferentes objetivos. En general, la estrategia de pruebas suele seguir las siguientes etapas:

- Las pruebas comienzan a nivel de módulo.
- Una vez terminadas las pruebas de los módulos, se prueba la integración del sistema completo y su instalación.

- Las pruebas terminan cuando el cliente acepta el producto y éste pasa a explotación.

En la figura 3.2 se muestra el modelo de ciclo de vida en V, relacionando los productos obtenidos durante el proceso de desarrollo y las fases de las pruebas. El modelo en V [3] es un proceso que representa la secuencia de pasos en el desarrollo del ciclo de vida de un proyecto. Describe las actividades y resultados que han de ser producidos durante el desarrollo del producto. La parte izquierda de la v representa la descomposición de los requisitos y la creación de las especificaciones del sistema. El lado derecho de la V representa la integración de partes y su verificación.

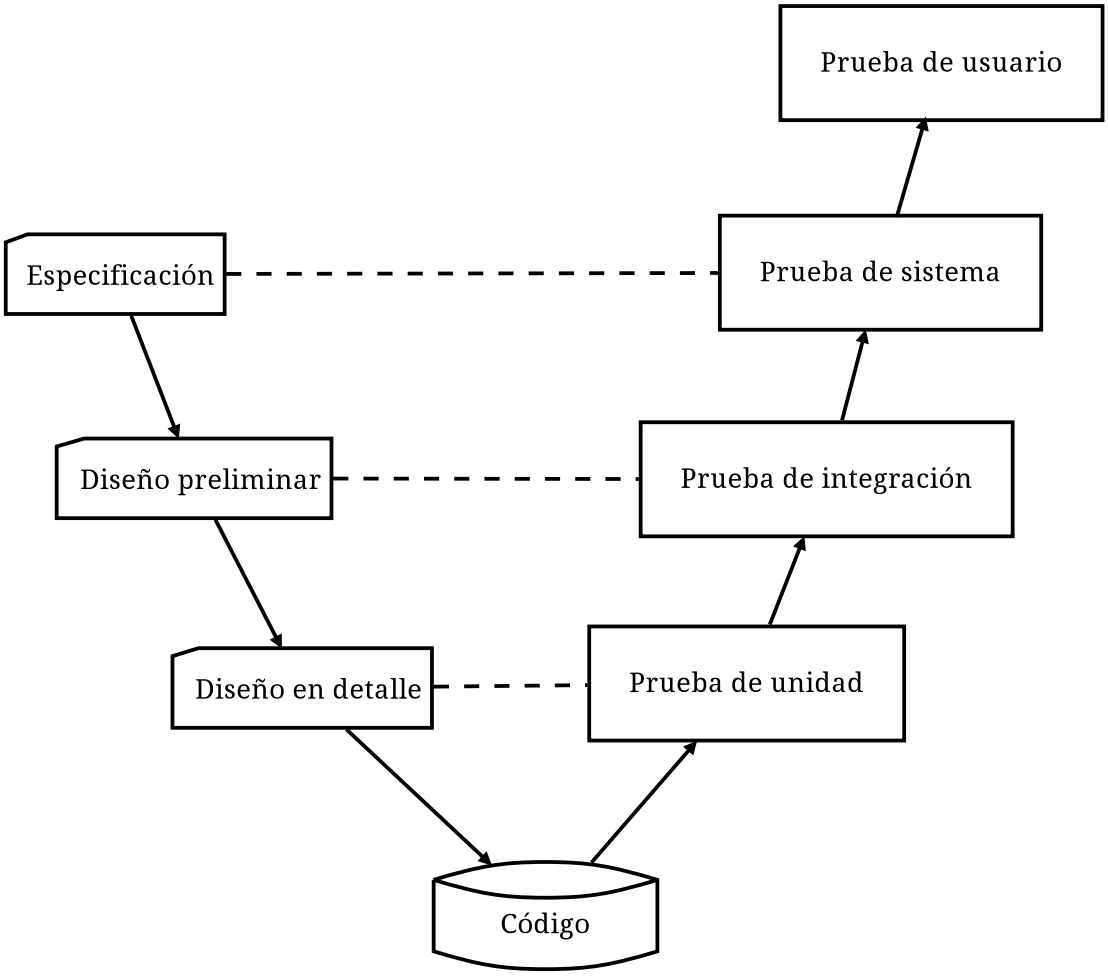

Figura 3.2: Modelo en V: relación entre productos de desarrollo y fases de pruebas

A continuación se detallan algunos niveles de prueba de software.

3.3.1. Pruebas de unidad

La prueba de unidad, o prueba de módulo, se concentra en el esfuerzo de verificación de la unidad más pequeña del diseño del software: el componente o módulo de software. Es decir, se prueban importantes caminos de control para descubrir errores dentro de los límites del módulo.

Las pruebas de unidad se concentran en la lógica del procesamiento interno y en las estructuras de datos dentro de los límites de un componente. Este tipo de pruebas se puede aplicar en paralelo a varios componentes.

La prueba de unidad suele considerarse adyacente al paso de la codificación. El diseño de las pruebas de unidad puede realizarse antes de que empiece la codificación o después de que se haya generado el código fuente.

3.3.2. Pruebas de integración

Estas pruebas se utilizan para probar la integración de los módulos teniendo en cuenta la estructura del sistema.

El programa se construye y se prueba en pequeños incrementos, en los cuales resulta más fácil aislar y corregir los errores. Es más probable que se prueben por completo las interfaces y se vuelve factible la aplicación de un enfoque de prueba sistemática.

La integración puede ser de los siguientes tipos:

Integración incremental o ascendente Se integra el módulo que se va a probar con el conjunto de módulos ya probados. El número de módulos se incrementa progresivamente hasta formar el programa completo.

Integración no incremental o descendente Los módulos se integran al descender por la jerarquía de control, empezando con el módulo de control principal (programa principal). Se prueba cada módulo por separado, luego, se integran todos los módulos a la vez y se prueba el programa completo.

3.3.3. Pruebas de regresión

Cada vez que se agrega un nuevo módulo como parte de una prueba de integración, el software cambia, ocurren nuevas entradas y salidas, se invoca una nueva lógica de control, etc. Estos cambios llegan a causar problemas con funciones que antes funcionaban bien.

Por este motivo las pruebas de regresión se emplean para la ejecución repetida de casos de prueba, y de esta manera poder encontrar errores de depuración en software de actualización, que ya ha pasado todas las pruebas.

3.3.4. Pruebas del sistema

Es el proceso de probar el sistema completo y final, consiste en probar la integración de todos los elementos del sistema (hardware y software):

- Probar que el sistema cumple todos los requisitos funcionales considerando el sistema completo.
- Comprobar el funcionamiento y rendimiento de las interfaces hardware, software, de usuario y de operador.
- Probar la adecuación de la documentación de usuario.
- Verificar la ejecución y rendimiento en condiciones límites y de sobrecarga.

3.4. Técnicas de prueba del software

Dado que no se pueden probar todas las posibilidades de funcionamiento del software, la idea fundamental para el diseño de casos de prueba consiste en elegir aquellas posibilidades que, por sus características, se consideren representativas del resto. De esta forma se asume que, si no se detectan defectos en el software al ejecutar dichos casos, podemos tener cierto nivel de confianza (que dependerá de la elección de los casos) en que el programa no tiene defectos. La dificultad de esta idea está en saber elegir los casos que se deben ejecutar. Por tanto, una elección puramente aleatoria no

proporciona demasiada confianza en que se puedan detectar todos los defectos existentes.

Existen tres enfoques principales para el diseño de casos de prueba: el enfoque estructural o de caja blanca, el enfoque funcional o de caja negra y el enfoque aleatorio (basado en modelos estadísticos).

3.4.1. Pruebas de caja blanca

Constituye el llamado enfoque estructural, y básicamente consisten en centrarse en la estructura interna (implementación) del programa para elegir los casos de prueba (ver figura 3.3). Al emplear los métodos de prueba de caja blanca, el programador podrá derivar casos de prueba que:

1. Garanticen que todas las rutas independientes dentro del módulo se han ejercitado por lo menos una vez.

2. Ejerciten los lados verdadero y falso de todas las decisiones lógicas.

3. Ejecuten todos los bucles en sus límites y dentro de sus límites operacionales.

4. Ejerciten estructuras de datos internos para asegurar su validez.

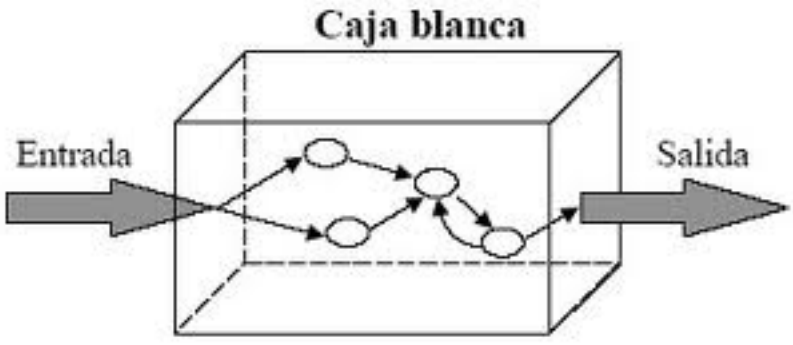

Figura 3.3: Pruebas de caja blanca

3.4.2. Pruebas de caja negra

Las pruebas de caja negra, también denominadas, pruebas de comportamiento, se concentran en los requisitos funcionales del software (ver figura 3.4). Es decir, permiten al programador derivar conjuntos de condiciones de entrada que ejercitarán por completo todos los requisitos funcionales de un programa, sin preocuparse de lo que pueda estar haciendo el módulo por dentro. La prueba exhaustiva del software consiste en probar todas las posibles entradas al programa.

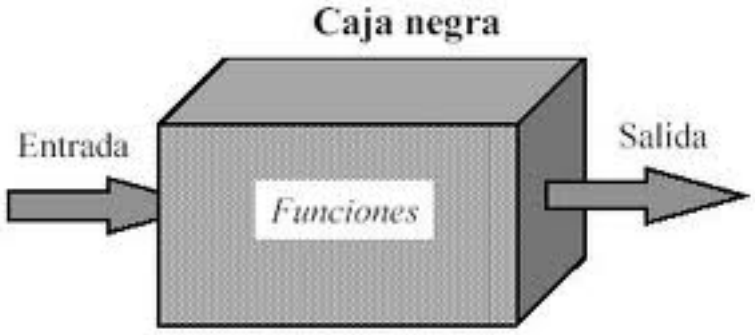

Figura 3.4: Pruebas de caja negra

Las pruebas de caja negra tratan de encontrar errores en las siguientes categorías:

1. Funciones incorrectas o faltantes.
2. Errores de interfaz.
3. Errores en estructuras de datos o en acceso a bases de datos externas.
4. Errores de comportamiento o desempeño.
5. Errores de inicialización y término.

Cabe mencionar que la prueba de caja negra no reemplaza a las pruebas de caja blanca, es, en cambio, un enfoque complementario que tiene probabilidades de descubrir una clase diferente de errores de los que se descubrirían con los métodos de caja blanca.

3.4.3. Pruebas aleatorias

Estas pruebas consisten en utilizar modelos (en muchas ocasiones estadísticos) que representan las posibles entradas al programa para crear a partir de ellos los casos de prueba. La prueba exhaustiva del software consiste en probar todas las posibles entradas al programa.

En estas pruebas se simula la entrada habitual del programa creando datos de entrada en la secuencia y con la frecuencia con las que podrían aparecer en la práctica (de manera repetitiva). Para ello habitualmente se utilizan generadores automáticos de casos de prueba.

3.5. Pruebas de mutaciones

Ya se han explicado las nociones más importantes sobre las pruebas software. En este apartado nos centraremos en una de las líneas de investigación que sigue el grupo UCASE: las pruebas de mutaciones.

La prueba de mutaciones es una técnica de prueba de caja blanca basada en fallos que consiste en introducir errores simples en el programa original mediante la aplicación de operadores de mutación. Los programas resultantes reciben el nombre de mutantes, que contienen pequeños cambios sintácticos con respecto al programa original [5]. Lo que pretenden este tipo de pruebas es medir la calidad de un conjunto de casos de prueba.

Cada operador de mutación se corresponde con una categoría de error típico que el desarrollador podría cometer. Así, si un programa contiene la instrucción $x > 1$ y disponemos de un operador de mutación que actúa sobre los operadores relacionales (cambia un operador relacional por otro), se podrían producir los mutantes que contengan: $x < 1$, $x = 1$, etc. [6]

Si un caso de prueba es capaz de distinguir al programa original del mutante, es decir, la salida del mutante y la del programa original son diferentes, se dice que mata al mutante. Por el contrario, se dice que un mutante es vivo para el conjunto de casos de prueba original, si ninguno de los casos de prueba es capaz de diferenciar al mutante del programa original, es decir, la salida del mutante y del programa original es la

misma. Por último, los mutantes erróneos son aquéllos que no pueden ser ejecutados al no ser un programa válido según las reglas del lenguaje.

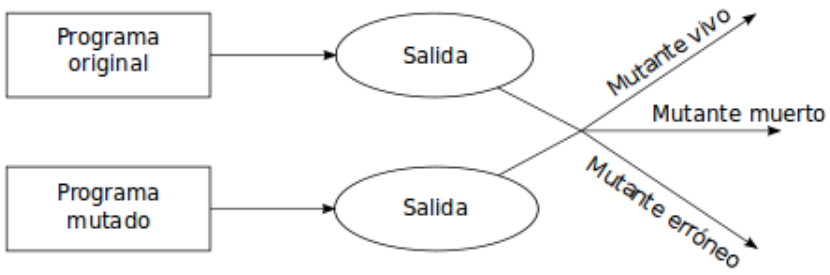

Figura 3.5: Esquema del funcionamiento de las pruebas de mutaciones

Una dificultad al aplicar las pruebas de mutación es la existencia de mutantes equivalentes. Dichos mutantes muestran el mismo comportamiento que el programa original, por lo que siempre dan las mismas salidas. La identificación de los mutantes equivalentes se suele realizar manualmente, siendo una tarea costosa.

No se deben confundir los mutantes equivalentes con los persistentes, que son aquellos que se producen cuando el caso de prueba no es lo bastante bueno para detectar el cambio. Si un mutante es persistente, lo usaremos para mejorar las pruebas con un nuevo caso de prueba que lo mate específicamente.

3.6. GAmera

GAmera es una herramienta desarrollada por el grupo UCASE para la generación y ejecución automática de mutantes para composiciones de Servicios Web en WS-BPEL. Es un algoritmo genético que busca conseguir rápidamente muchos mutantes persistentes, de forma que se puedan usar para mejorar el conjunto de casos de prueba.

GAmera incorpora un mecanismo de optimización que permite seleccionar un subconjunto de los mutantes totales que pueden generarse. Esto se logra mediante la utilización de un algoritmo genético que genera y selecciona sólo los mutantes de mayor calidad, reduciendo el coste computacional que implicaría la ejecución de todos los mutantes. Los resultados que proporciona esta herramienta permiten mejorar la calidad de los casos de prueba.

3.6.1. Estructura de GAmera

A continuación se detallan los tres componentes principales de GAmera (ver figura 3.6):

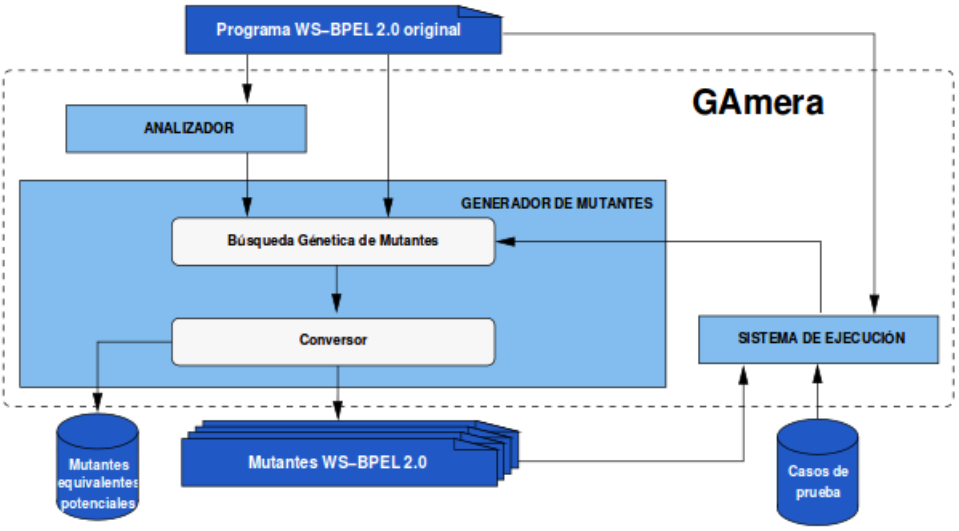

Figura 3.6: Estructura de GAmera

3.6.1.1. El analizador

Antes de que podamos generar los mutantes, es necesario identificar los elementos del código original que pueden ser mutados: de esto se encarga la fase de análisis.

El analizador de GAmera es el primer componente de la herramienta que actúa. Recibe como entrada la composición WS-BPEL a probar y determina los operadores de mutación que se le pueden aplicar.

Según el programa, algunos operadores se podrán usar y otros no. En el caso de que un operador no se pueda aplicar, el analizador nos informará de que dicho operador es aplicable a 0 instrucciones.

3.6.1.2. Generador de mutantes

Es el siguiente componente que entra en acción, partiendo de la información o que se recibe del analizador. La herramienta nos da la posibilidad de generar todos los

mutantes posibles, o bien un subconjunto de éstos que va a ser seleccionado por el algoritmo genético.

En este último caso, se llamará al componente denominado generador de mutantes, que está compuesto por dos elementos. El primero, denominado *Búsqueda Genética de Mutantes*, es un algoritmo genético en el que cada individuo representa a un mutante, capaz de generar y seleccionar de forma automática un conjunto de mutantes. Esta selección se realiza aplicando una función de aptitud que mide su calidad en función de si hay o no casos de prueba que lo matan. El segundo elemento es el *Conversor*, que transforma un individuo del algoritmo genético en un mutante WS-BPEL. Para realizar esta conversión, se utilizan hojas de estilos XSLT, una por cada operador de mutación.

3.6.1.3. Sistema de ejecución

En esta fase se ejecutan tanto el fichero original como cada uno de los ficheros mutados contra un conjunto de casos de prueba.

Los resultados de las ejecuciones de los mutantes se comparan con los resultados de la ejecución del código original. Si para una determinada prueba, un mutante ha dado un resultado distinto al resultado que ha dado el código original, marcamos dicha prueba con un 1. Si el resultado es igual, la marcamos con un 0. Si la ejecución es errónea, la marcamos con un 2.

Como resultado obtenemos un vector de 0, 1 y 2 por cada mutante. Si algún elemento del vector es 1, el mutante está muerto. Si todos son 0, está vivo. Cuando la ejecución es errónea, todo el vector estará a 2.

Al tener un vector por cada mutante lo que realmente conseguimos es una matriz llamada matriz de ejecución, que nos indica para cada mutante y cada prueba, el resultado obtenido.

3.7. Un ejemplo con MuBPEL

MuBPEL es el componente específico para el manejo de mutantes WS-BPEL en GAmera, con la diferencia de que GAmera utiliza el algoritmo genético y MuBPEL se limita a obtener todos los mutantes posibles al aplicarle al programa original los operadores

de mutación.

A continuación mostraremos un ejemplo completo de aplicación de pruebas de mutación utilizando la herramienta MuBPEL, la cual se ejecuta por línea de comandos.

Recordemos que el primer paso es el de análisis, con lo cual utilizamos esta utilidad integrada en MuBPEL mediante la siguiente orden:

```
mubpel analyze LoanApprovalProcess.bpel
```

Obtendremos como resultado la lista de operadores, en la que se muestra el nombre de cada operador, el número de localizaciones en las que se puede aplicar y el número de atributos disponibles:

```
ISV 0 1
EAA 0 4
EEU 0 1
ERR 2 5
ELL 0 1
ECC 13 1
ECN 1 4
EMD 0 2
EMF 0 1
AFP 0 1
ASF 3 1
AIS 0 1
AIE 2 1
AWR 0 1
AJC 0 1
ASI 12 1
APM 0 1
APA 0 1
XMF 0 1
XMC 0 1
XMT 0 1
XTF 0 1
XER 0 1
XEE 0 1
AEL 17 1
EIU 2 1
EIN 2 1
EAP 2 1
EAN 2 1
```

```
CFA 17 1
CDE 2 2
CCO 2 2
CDC 2 2
```

En este caso el primer operador aplicable es ERR, el que ocupa el cuarto lugar. Este operador sustituye un operador relacional (=, ! =, <, >, <=, >=) por otro del mismo tipo. Se puede aplicar en 2 localizaciones con 5 valores distintos. Crearemos un mutante aplicándolo sobre el programa original:

```
mubpel apply LoanApprovalProcess.bpel 4 2 5 > mutante1.bpel
```

Se habrá generado un fichero llamado mutante1.bpel conteniendo el programa mutado. A continuación, ejecutaremos las pruebas unitarias sobre el programa original, utilizando BPELUnit. De esta operación obtenemos un fichero XML con los resultados de las pruebas:

```
mubpel run LoanApprovalProcess−Velocity.bpts LoanApprovalProcess.bpel > resultado.xml
```

Por último, ejecutaremos las pruebas sobre el mutante, y compararemos los resultados:

```
mubpel comparefull LoanApprovalProcess−Velocity.bpts LoanApprovalProcess.bpel resultado.xml mutante1.bpel
```

La orden `comparefull` realiza ambas acciones, ejecutar las pruebas al mutante y comparar los resultados obtenidos con los del programa original. Tras ejecutar esta orden obtenemos la comparación de los resultados de las pruebas unitarias sobre el programa mutado:

```
mutante1.bpel 0 0 1 1 1 1 1 1 1
```

Podemos observar que las dos primeras pruebas devuelven 0, luego ambos programas obtienen la misma salida y no han notado el cambio. Y las pruebas restantes obtienen 1, con lo que obtienen salidas distintas, es decir, se ha detectado el cambio. Al haber al menos una prueba que ha detectado la modificación, el mutante está muerto.

Si el vector fuera todo ceros, podríamos concluir que el conjunto de casos de prueba es insuficiente para detectar al mutante.

Capítulo 4

Análisis del proyecto

En este capítulo hablaremos de las necesidades que debe cumplir el proyecto, realizando un análisis de la aplicación.

4.1. Requisitos funcionales

Los requisitos funcionales son los que describen las funciones del sistema.

Como ya se ha comentado anteriormente, la herramienta ServiceAnalyzer se creó para analizar Servicios Web, y generaba plantillas parametrizadas para producir mensajes de acuerdo a todas las restricciones impuestas desde WSDL, XML Schema y el WS-I Basic Profile 1.1.

En TestGenerator implementa un generador aleatorio de los datos que necesitaban esas plantillas, datos almacenados en el *vm*. Además, implementa un lenguaje de dominio específico (ficheros *spec*), capaz de representar el conjunto de datos y restricciones impuestas por ServiceAnalyzer.

Y finalmente, SpecGenerator extrae la información necesaria del catálogo generado por ServiceAnalyzer con la finalidad de crear un *spec*, para una cierta operación, sentido y servicio.

El presente proyecto se crea con el fin de realizar un análisis a una composición WS-BPEL para poder realizar pruebas a la misma. Para cumplir este objetivo, se hará uso de las herramientas anteriormente citadas.

Se definen, pues, los requisitos funcionales del sistema:

- Analizar la composición WS-BPEL.
- Analizar las plantillas generadas por ServiceAnalyzer de los servicios involucrados en la composición.
- Analizar los distintos *spec* de estos mismos servicios.
- Generar un único *spec* con los datos y restricciones.
- Generar el fichero *vm*.
- Generar el fichero *bpts*.

4.2. Análisis de las composiciones WS-BPEL

Una vez se han definido todas las tecnologías necesarias para abordar el problema que pretende solucionar este PFC, nos vamos a centrar en el análisis necesario que hay que realizar a las composiciones WS-BPEL para crear el *bpts* que las pruebe.

Ya hemos hablado de las composiciones WS-BPEL y sus Servicios Web definidos en los ficheros WSDL, sin embargo, es necesario detallar alguna información de estos ficheros para entender el funcionamiento de las pruebas usando BPELUnit.

4.2.1. *Partners* de un proceso WS-BPEL

Un proceso WS-BPEL pretende componer servicios, y para conseguirlo, estos servicios deben ser identificados e invocados. Estos servicios deben estar descritos en los ficheros WSDL, al igual que el proceso WS-BPEL que no es más que un servicio en sí mismo.

Un Servicio Web que interactúa con el proceso WS-BPEL de alguna manera se denomina *partner*. La relación que tiene un proceso WS-BPEL con un *partner* puede tomar tres formas:

- El *partner* invoca el proceso WS-BPEL.
- El *partner* es invocado por el proceso WS-BPEL.

- Ambos: el *partner* es invocado por el proceso WS-BPEL e invoca al proceso WS-BPEL.

La relación entre un *partner* y el proceso WS-BPEL se describe de manera abstracta, definiendo los *roles* que pueden tomar el proceso o el *partner*. Estas relaciones se describen, normalmente, en el fichero WSDL asociado al servicio, bajo la etiqueta `partnerLinkType`. Un `partnerLinkType` contiene al menos un *rol* y dos como mucho.

Las secciones que contienen las definiciones de los *partner links*, los cuales están basados en `partnerLinkType`, a las que se les añade la información de qué *rol* va a tener en la composición, se denominan `partnerLink` y están definidas en la propia composición WS-BPEL.

Listado 4.1: Definiciones *partnerLinks*

```
<partnerLinks>
   <partnerLink name="assessor"
            partnerLinkType="ns2:AssessorService1"
            partnerRole="AssessorServicePortTypeRole"/>
   <partnerLink name="approver"
            partnerLinkType="ns1:ApprovalService1"
            partnerRole="ApprovalServicePortTypeRole"/>
   <partnerLink name="client" partnerLinkType="ns3:LoanService1"
            myRole="LoanServicePortTypeRole"/>
</partnerLinks>
```

La definición `myRole` la tienen aquellos procesos que actúan como clientes de la composición (los que llaman al proceso WS-BPEL), y `partnerRole` indica, como su nombre indica, el *rol* del *partner*. Si un `partnerLink` tiene ambos *roles* definidos, quiere decir que es una comunicación asíncrona en la que primeramente tomará el *rol* de cliente, llamando al proceso, y en la respuesta asíncrona describirá su *rol* con el `partnerRole`.

4.2.2. La lógica de negocio

La lógica de negocio de una composición WS-BPEL consiste en una serie de actividades que son ejecutadas secuencialmente o en paralelo. Las actividades pueden ser

básicas o estructuradas. Las actividades básicas incluyen la invocación de otros Servicios Web o la llegada de invocaciones entrantes; las estructuradas se usan para definir las relaciones entre actividades.

Las actividades más importantes en los procesos WS-BPEL (ver 4.2) reciben y envían mensajes a y desde los *partners*. Tres actividades son las encargadas de ejecutar este comportamiento:

- La actividad `<receive>` espera el mensaje de un *partner*.
- La actividad `<invoke>` invoca a un *partner*.
- La actividad `<reply>` envía una respuesta al mensaje previamente recibido por la actividad `<receive>`.

Listado 4.2: Actividades básicas de una composición WS-BPEL

```
<receive name="Receive1" createInstance="yes" partnerLink="client"
          operation="grantLoan" portType="ns3:LoanServicePortType"
          variable="processInput"/>
...
<invoke name="queryAssessor" partnerLink="assessor"
            operation="assessLoan" portType="ns2:AssessorServicePortType"
            inputVariable="assessorInput"
            outputVariable="assessorOutput"/>
...
<reply name="Reply1" partnerLink="client" operation="grantLoan"
        portType="ns3:LoanServicePortType" variable="processOutput"/>
```

Los atributos que se definen en estas actividades son los siguientes:

partnerLink Contiene el nombre del *partner link* usado. El motor WS-BPEL usa este nombre para identificar el destino del actual *partner* en tiempo de ejecución.

porType Este valor define el tipo de puerto del servicio objetivo que contiene la operación a invocar. Está descrito en el fichero WSDL.

operation Define la operación a invocar, también definido en el WSDL.

inputVariable Contiene la información a ser enviada al servicio.

outputVariable Es inicializado con la respuesta del *partner*.

Otra actividad que hay que mencionar, ya que interviene en cierta manera en la lógica, es la actividad `<pick>`. Esta actividad es un conjunto de ramas de la forma evento/actividad, cada una puede tomar alguno de dos tipos distintos: ramas `<onMessage>` y ramas `<onAlarm>`. Cuando ocurre alguno de estos desencadenadores, se ejecuta la actividad asociada a cada uno de ellos. Esta actividad debe incluir por lo menos un desencadenador `<onMessage>`.

Si la actividad `<pick>` tiene el atributo `createInstance` a "yes", las actividades `<onMessage>` que estén dentro de esta actividad `<pick>` se comportarán de la misma manera que una actividad `<receive>`.

Listado 4.3: Actividad `<pick>`

```
1 <pick createInstance="yes" name="FirstMessage">
2   <onMessage operation="submit"
3            partnerLink="seller"
4            portType="tns:sellerPT"
5            variable="sellerInfo">
6 ...
7 </pick>
```

Se puede invocar varias veces al mismo servicio a lo largo de la composición, con las mismas variables de entrada y de salida. Sin embargo, por falta de tiempo, y por la complejidad de extraer la lógica de la composición, en esta primera versión de BPTS-Generator se restringe esta opción y se podrá invocar el mismo servicio varas veces pero las variables de entrada o salida deberán ser distintas. En la sección de trabajo futuro se detallará la necesidad de realizar un análisis más exhaustivo de la composición, para detectar todos los posibles caminos de la misma, y realizar unas pruebas más detalladas.

4.2.3. Conjuntos de correlación

Un proceso WS-BPEL especifica la plantilla de un proceso de negocio, donde múltiples instancias pueden correr al mismo tiempo. Estas instancias son completamente independientes una de otras.

Debido a que pueden existir varias instancias que pueden estar esperando respuesta de un Servicio Web, el servidor WS-BPEL debe estar en condiciones de correlacionar las respuestas del Servicio Web con la instancia del proceso correcta.

Cuando el motor WS-BPEL recibe un mensaje busca el proceso que puede manejarlo. Para realizar esta búsqueda utiliza el conjunto de correlación que contiene los datos necesarios para realizar la correspondencia entre el mensaje y el proceso que lo está esperando.

Ha sido fundamental analizar este detalle de las composiciones, ya que si se va a probar una composición hay que tener en cuenta la correlación de los mensajes entre los distintos servicios.

En WS-BPEL, la correlación está basada en dos conceptos: *propiedades* y los *conjuntos de correlación*.

- Una propiedad tiene un nombre y un tipo, e indica las partes de los mensajes que deben enviarse o recibirse con el mismo ID.
- Un conjunto de correlación consiste en múltiples propiedades y representa el ID de la conversación entre las instancias.

Los conjuntos de correlación permite al diseñador de la composición decidir qué parte de los mensajes se envían y se reciben de los *partners*, identificándolos con un mismo ID.

Las propiedades se definen en los ficheros WSDL. Una propiedad se define primero con un nombre y un tipo con la etiqueta `<property>`, y se liga a varias partes de mensajes bajo la etiqueta `<propertyAlias>`. Todos los `<propertyAlias>` con el mismo `propertyName` delimitan un conjunto de correlación, debiendo de coincidir los `part` y/o `query` de los `messageType`. Veamos el siguiente ejemplo:

Listado 4.4: Ejemplo de `<property>` y `propertyAlias`

```
1 <message name="sellerInfoMessage">
2   <part name="inventoryItem" type="xsd:string"/>
3   <part name="askingPrice" type="xsd:integer"/>
4 </message>
5
6 <message name="buyerInfoMessage">
```

```
7    <part name="item" type="xsd:string"/>
8    <part name="offer" type="xsd:integer"/>
9   </message>
10  ...
11  <bpws:property name="negotiatedItem" type="xsd:string"/>
12
13  <bpws:propertyAlias propertyName="tns:negotiatedItem"
14                      messageType="tns:sellerInfoMessage"
15                      part="inventoryItem"
16                      query="/inventoryItem"/>
17
18  <bpws:propertyAlias propertyName="tns:negotiatedItem"
19                      messageType="tns:buyerInfoMessage"
20                      part="item"
21                      query="/item"/>
```

En este caso se define una propiedad llamada `negotiatedItem` de tipo cadena. A continuación se describen dos mensajes que deben coincidir con el mismo ID. Los mensajes son `sellerInfoMessage` y `buyerInfoMessage`, las partes `inventoryItem` y `item` respectivamente. Es decir, estas partes de estos mensajes deben coincidir para hacer corresponder los mensajes entre las instancias.

En la composición se definen los conjuntos de correlación mediante la etiqueta `<correlationSets>` y se declaran cuando se llama a una actividad de envío/recibo de mensajes:

Listado 4.5: Ubicación de los conjuntos de correlación en la composición

```
1   <correlationSets>
2       <correlationSet name="negotiationIdentifier" properties="tns:negotiatedItem"/>
3   </correlationSets>
4   ...
5   <receive createInstance="yes" name="SellerReceive" operation="submit" partnerLink="
        seller" portType="tns:sellerPT" variable="sellerInfo">
6     <correlations>
7       <correlation set="negotiationIdentifier" initiate="join"/>
8     </correlations>
9   </receive>
```

4.3. Creación de un *bpts*

Tanto el cliente como todos los *partners* son simulados por el framework de BPELUnit. El programador que desee realizar pruebas a una composición WS-BPEL tendrá que especificar qué dato del cliente y de los *partners* se deberían esperar del proceso BPEL y enviarlos de vuelta. BPELUnit permite, entre otras cosas, lo siguiente:

- Especificar las llamadas síncronas y asíncronas de un proceso BPEL en pruebas; BPELUnit incluye una implementación de direccionamiento de Servicios Web que crea las llamadas tanto del lado del cliente como del servidor.

- Especificaciones de los datos literales en XML, soportado por la codificación *document/literal* y *rpc/literal*, con la posibilidad de añadir más codificaciones vía un punto de extensión. La diferencia entre el estilo *document* y el *rpc* es que en en *rpc* cada parte se corresponde con un tipo XML Schema y la correspondencia entre el mensaje entrante y la operación se hace mediante un envoltorio con el nombre de la operación, meintras que en *document* cada parte se corresponde con un elemento XML Schema y la correspondencia entre el mensaje entrante y la operación se hace mediante el propio elemento (no hay envoltorio).

- Permite expresiones XPath.

Las expresiones XPath se pueden usar tanto para comprobar condiciones sobre los mensajes enviados por la composición, como para calcular retrasos a introducir en los envíos y activar y desactivar actividades y *partner tracks* en función de los valores de las variables.

En la figura 4.1 se muestra la simulación que realiza BPELUnit de una composición WS-BPEL.

Recordemos que un *mockup* es un servicio sustituido de un servicio real en una simulación, que implementa un comportamiento predefinido, y que se emplea para comprobar si la salida de un programa para un caso de prueba es la esperada.

El servicio que actúa como cliente se corresponde con la etiqueta `<clientTrack>` del *b*pts; y el resto de *partners* se corresponden con los `<partnerTrack>`.

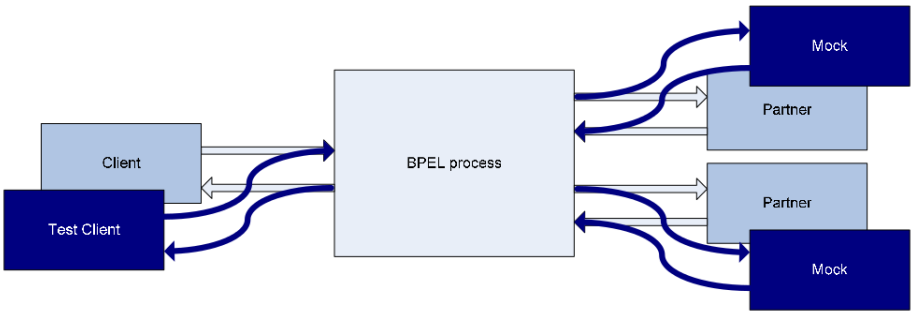

Figura 4.1: Simulación de BPELUnit

Puesto que se analizan composiciones síncronas (la comunicación no se interrumpe), el bloque que le corresponde al `<clientTrack>` es necesariamente el bloque `<sendReceive>`, que recordemos que envía un mensaje, espera una respuesta síncrona, y verifica la respuesta.

Por otro lado a los `<partnerTrack>` les corresponde el bloque `<receiveSend>`, es decir lo opuesto al cliente, ya que primero esperan el mensaje del cliente, lo verifica y envía de vuelta una respuesta síncrona.

Ambas actividades tienen un apartado `<send>` y otro `<receive>`, en los que se envía y recibe la información de los mensajes de los servicios. En esta primera versión de BPTSGenerator solamente se añadirá la plantilla Velocity al apartado `<send>`, ya que extraer de la composición el contenido de lo que devuelven los mensajes es muy complejo y sería necesario aplicar razonamiento automático[1], lo que se escapa del objetivo del presente proyecto.

La secuencia que sigue BPELUnit para ejecutar un caso de prueba a una composición WS-BPEL es la descrita en la figura 4.2 [19]. Se trata de una interacción entre el cliente, los *partner tracks* y el PUT (proceso que se está probando o Process Under Test en inglés).

El cliente realiza una solicitud síncrona al proceso PUT, el cual se encarga de mantener la conversación entre los dos *partners*, de los que se puede observar que uno es invocado síncronamente y el otro asíncronamente.

[1]Habría que implementar un "oráculo de prueba", es decir, algo que mire la salida de la prueba y diga si se ha superado o no.

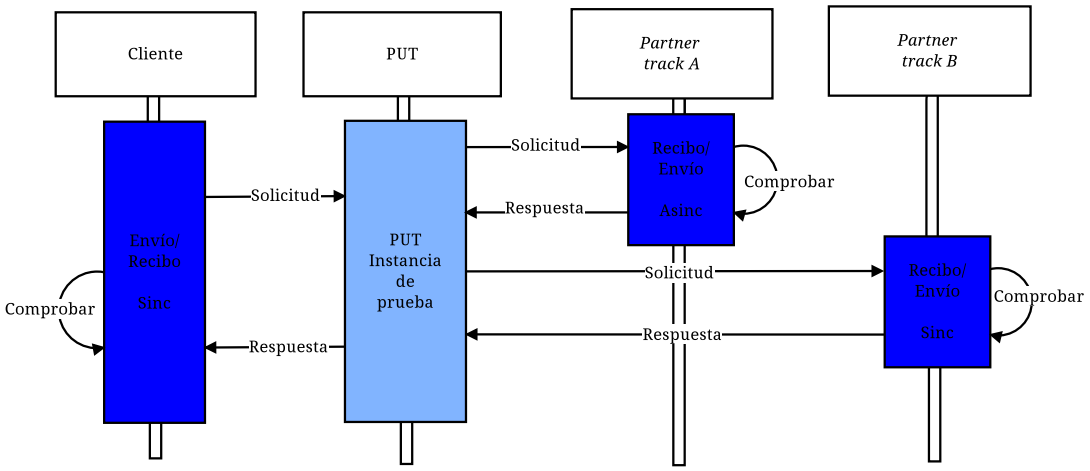

Figura 4.2: Secuencia de un caso de prueba

4.4. Requisitos de implementación

Los requisitos de implementación impuestos por el grupo UCASE son los siguientes:

- El lenguaje utilizado para la implementación del proyecto deberá ser Java [8] ya que la mayoría de las aplicaciones desarrolladas por el grupo están escritas en este lenguaje, por lo que de esta manera se facilita la reutilización de código.

- Se exige la realización de pruebas unitarias para poder evaluar el funcionamiento del sistema. Para ello se utilizará el framework JUnit.

- El grupo exige también utilizar un entorno de integración continua, para poder llevar mejor el control de los distintos trabajos que se estén realizando.

4.5. Atributos del sistema

El sistema deberá cumplir las siguientes propiedades:

- Mantenibilidad: es fundamental para que posibles correcciones se hagan rápidas y en un futuro que este trabajo pueda ser ampliado.

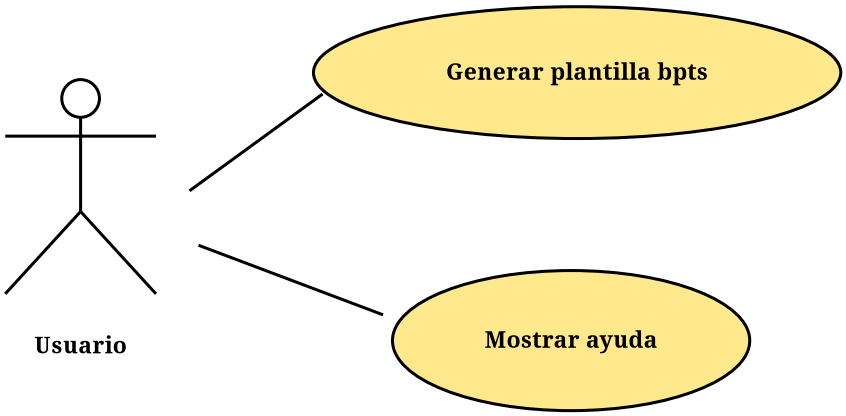

Figura 4.3: Diagrama de casos de uso

- Facilidad de uso: a pesar de que el proyecto está dirigido a personas con conocimientos informáticos elevados, se intentará facilitar su uso en la medida de lo posible para que no se haga demasiado tedioso trabajar con la aplicación.
- Licencia libre: para que la aplicación tenga el mejor uso público posible se le otorga dicha licencia.

4.6. Casos de uso

Los casos de uso son una técnica de especificación de requisitos que podemos usar tanto en desarrollos estructurados como orientados a objetos; aunque en el caso de los orientados a objetos es particularmente útil, ya que será una referencia a lo largo de todo el ciclo de vida. En esta etapa se identifican:

- Actores.
- Casos de uso.
- Descripción de cada caso de uso.

En la figura 4.3 podemos ver representado el diagrama de casos de uso, utilizando la notación UML.

A continuación se detallan los contenidos de cada caso de uso.

4.6.1. Caso de uso: Generar plantilla *bpts*

Actor principal Usuario que desea crear una plantilla de pruebas *bpts*.

Precondición Existe un documento WS-BPEL proporcionado en la ruta especificada.

Postcondiciones Se muestra por pantalla el contenido del fichero *bpts*.

Escenario principal

1. El usuario introduce como argumento en la terminal la ruta del fichero WS-BPEL mediante la orden `bpts-generator (ruta)`.
2. El sistema genera la plantilla de pruebas.

Variaciones

1a. El nombre de la orden es incorrecto.

    1. El sistema muestra un mensaje indicando que no reconoce la orden y cancela el caso de uso.

1b. El número de argumentos introducido no es correcto.

    1. El sistema muestra un mensaje indicando que el número de argumentos introducido es incorrecto y muestra otro mensaje por pantalla el modo del uso de la herramienta. Se cancela el caso de uso.

2a. La definición del proceso BPEL no es válida.

    1. El sistema muestra el mensaje de error por pantalla y cancela el caso de uso.

4.6.2. Caso de uso: Mostrar ayuda

Actor principal Usuario que desea ver la ayuda de la herramienta.

Precondición Ninguna.

Postcondiciones Se muestra por pantalla la información correspondiente al nombre y la versión del programa.

Escenario principal

1. El usuario indica su intención de obtener la ayuda del programa.
2. El sistema muestra la ayuda por pantalla.

4.7. Modelo conceptual de datos

Este modelo describe las estructuras de datos de un sistema y las relaciones estáticas que existen entre ellos. Está orientado a representar los elementos que intervienen en un problema concreto y normalmente contienen:

- Clases de objetos.
- Asociaciones entre clases de objetos.
- Atributos de las clases de objetos.
- Restricciones de integridad.

En la figura 4.4 se muestra el diagrama conceptual para el sistema.

El sistema comenzará leyendo el fichero WS-BPEL recibido como argumento por la línea de órdenes. Se analizará la composición y se extraerán las actividades que tienen que intervenir en las pruebas, que serán las actividades `receive, onMessage` o `invoke`.

Cada actividad invoca un servicio, el cual viene descrito en un fichero WSDL. Con la herramienta ServiceAnalyzer podemos generar, a partir del fichero WSDL su correspondiente catálogo, el cual contendrá la plantilla Velocity, que tras ser analizada podrá incluirse en el *bpts*.

Por otra parte, una vez se dispone del catálogo de servicios, se crea el *spec* gracias a la herramienta SpecGenerator. Tras analizarse y unificarse con los *spec* de las demás actividades, crearemos el *vm* con TestGenerator.

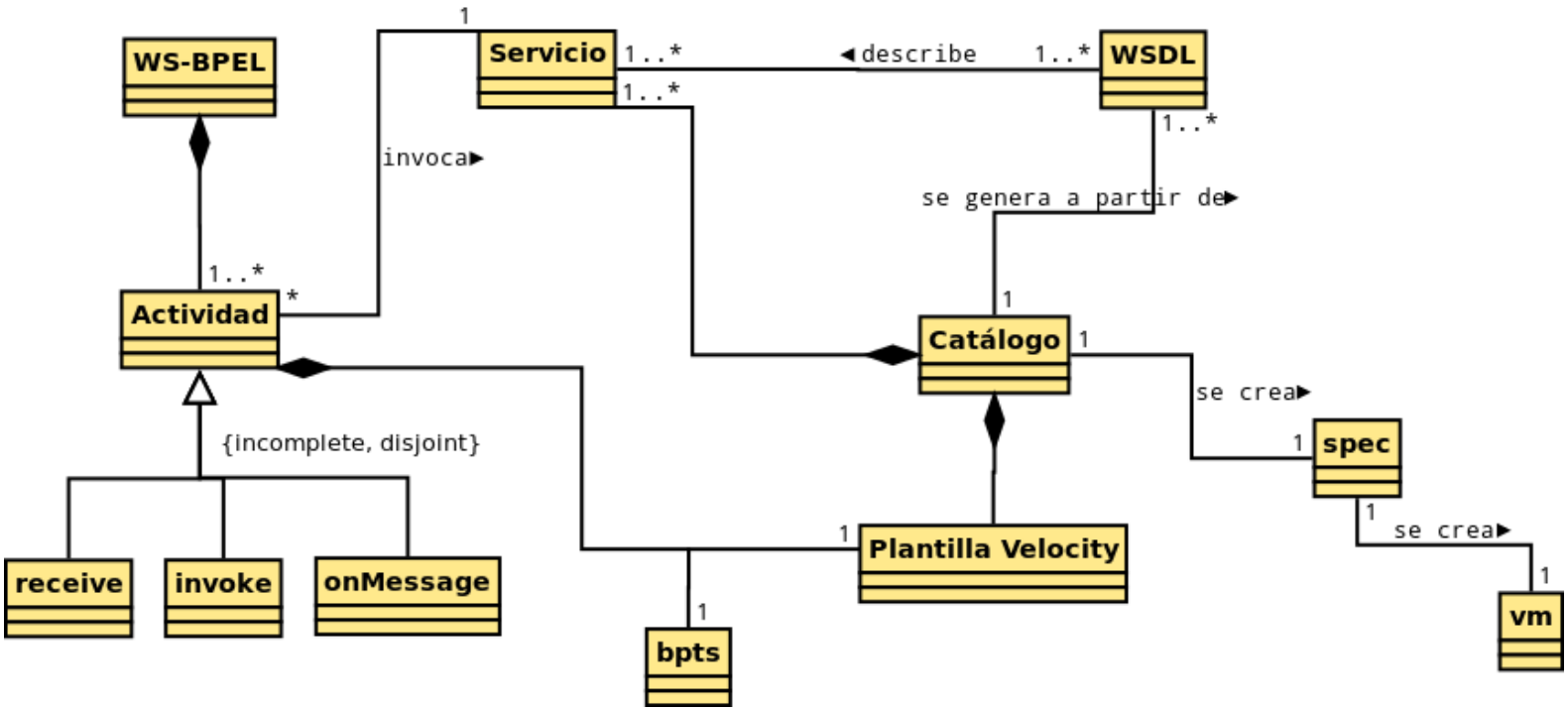

Figura 4.4: Modelo conceptual de datos del sistema

Capítulo 5

Diseño del proyecto

A continuación hablaremos de las condiciones que nos llevaron a tomar las decisiones de diseño en el proyecto y la estructura que el mismo seguirá.

5.1. Arquitectura del sistema

Desde la primera vez que un programa se dividió en módulos, los sistemas de software han tenido arquitecturas, y los programadores han sido responsables de las interacciones entre los módulos y las propiedades globales del ensamblaje. [27]

La arquitectura del software de un programa es la estructura o las estructuras del sistema, que incluyen los componentes del software, las propiedades visibles externamente de esos componentes y las relaciones entre ellos [1].

Un condicionante que ha determinado la arquitectura del sistema ha sido la búsqueda de la calidad del software: el objetivo es crear un software flexible, fácilmente mantenible, que permita la reutilización de sus componentes y que sea fiable.

Para este proyecto se ha elegido la **Arquitectura de llamada y retorno** [25]. Este estilo arquitectónico permite que un diseñador de software obtenga una estructura de programa que resulta relativamente fácil modificar y cambiar de tamaño. En esta categoría hay dos subestilos [1]:

Arquitectura de programa principal/subprograma Esta estructura de programa clásica separa la función en una jerarquía de control donde un programa "principal"

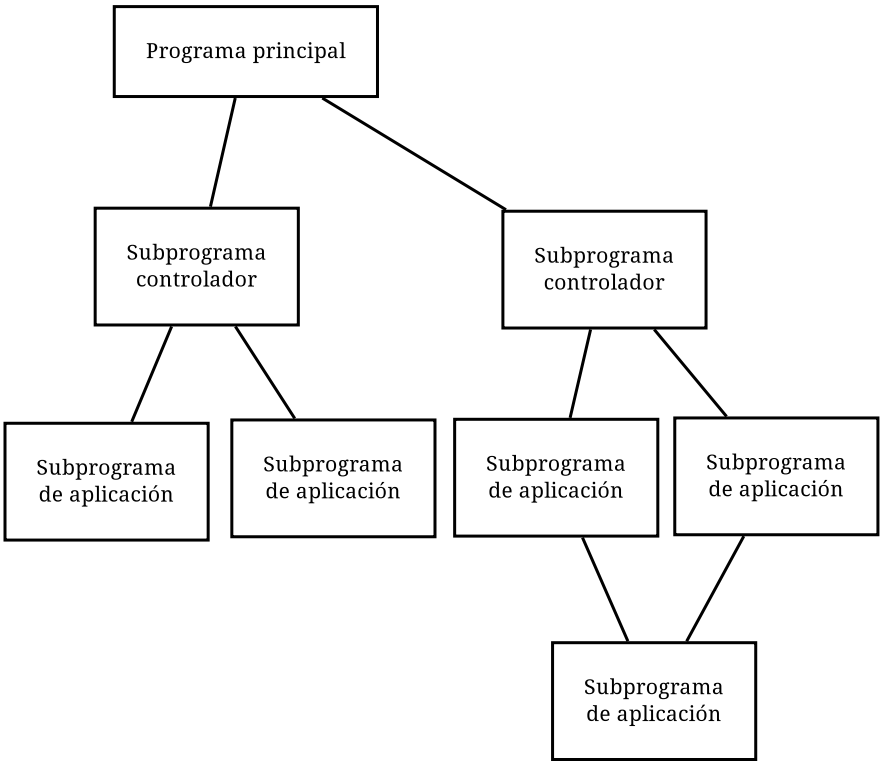

Figura 5.1: Arquitectura de programa principal/subprograma

invoca a varios componentes de programa, que a su vez pueden invocar a otros componentes. Podemos ver en la figura 5.1 ilustrada una arquitectura de este tipo.

Arquitectura de llamada de procedimiento remoto Los componentes de una arquitectura de programa principal/subprograma se distribuyen entre varias computadoras de una red.

De las dos categorías se ha elegido la **Arquitectura de programa principal/subprograma**, por los objetivos comentados anteriormente.

Este proyecto se encuentra dividido en dos grandes bloques (filtros). Esta división está basada en la funcionalidad que cumplen cada uno de los bloques. Los bloques son:

- Analizador.
- Generador.

La división en estos dos bloques nos aporta grandes ventajas. Entre ellas podemos destacar:

- Desarrollo: con esta estructura, es más sencillo y cómodo usar la metodología llevada en este proyecto que ha sido iterativa por prototipo.
- Corrección de errores: al estar claramente diferenciadas las funcionalidades, es más cómodo identificar problemas y corregir errores, asegurando que el resto de partes funcionen correctamente.
- Ampliación del sistema: si en algún momento necesitamos ampliar algún formateado extra o ampliar los tipos de ficheros de entrada al programa, gracias a dicha división es menos costoso y reduce la aparición de errores.
- Reutilización de código: puesto que el proyecto pertenece al grupo de investigación UCASE, es más sencilla la reutilización de alguna de sus partes en otros proyectos.

5.1.1. Analizador

Este bloque estructural del proyecto es el encargado de analizar los ficheros que recibe de entrada con el fin de crear una estructura de datos que guarde la información relevante para, en una etapa más tardía, poder generar los ficheros. El sistema es capaz de analizar composiciones WS-BPEL síncronas, y sus respectivos Servicios Web.

En la figura 5.2 podemos observar el diagrama de clases del analizador. Este bloque está implementado en la clase PARSER. Esta clase es la que comienza el análisis de la composición, seleccionando aquellas actividades que se probarán. Primeramente comprueba que no se repitan las actividades y para almacenar toda la información que nos interesa tener se implementó una clase que consiguiera este objetivo, la clase ACTIVITY.

Para el análisis de los ficheros WSDL utilizaremos ServiceAnalyzer. Dicho programa nos generará un catálogo que será el que analizaremos utilizando las clases del propio ServiceAnalyzer, las cuales utilizan la herramienta XMLBeans.

XMLBeans [9] es un proyecto que forma parte de la *Apache Software Foundation*, la cual es capaz de realizar automáticamente el *data binding* entre XML y Java, es decir,

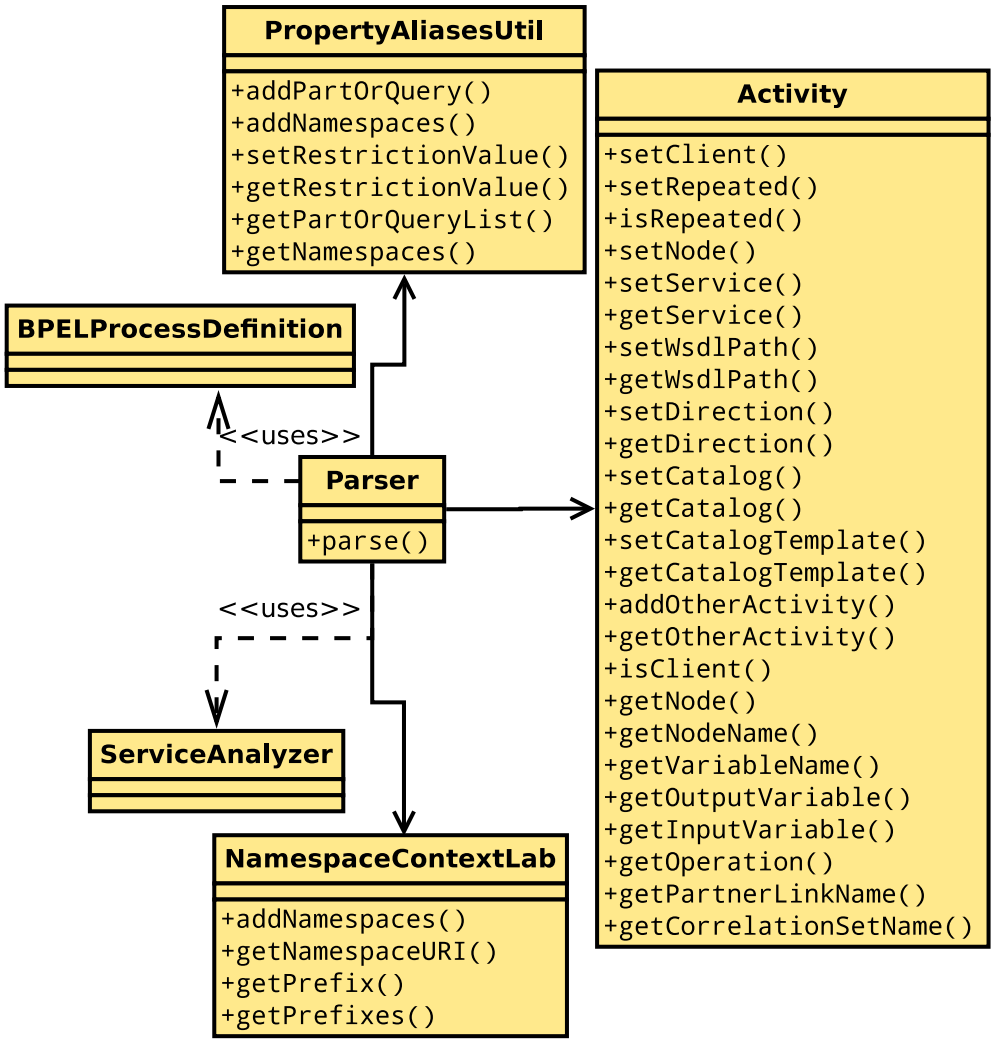

Figura 5.2: Diagrama del analizador

puede convertir tanto XML a Java como Java a XML. Esta herramienta es muy útil para manejar ficheros XML ya que vuelca toda la información del fichero XML en objetos Java fáciles de manipular, y también facilita la creación de ficheros XML desde código Java.

En el listado 5.1 tenemos un ejemplo del catálogo que genera ServiceAnalyzer de un fichero WSDL.

Listado 5.1: Ejemplo de un catálogo generado por ServiceAnalyzer

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

```
<mes:services xmlns:mes="http://serviceAnalyzer/messageCatalog">
  <mes:service name="LoanServiceService" uri="http://j2ee.netbeans.org/wsdl/
      LoanService">
    <mes:port name="LoanServicePort" address="http://localhost:8080/active-bpel/
        services/LoanServiceService">
      <mes:operation name="grantLoan">
        <mes:input>
          <mes:decls>
            <mes:variable name="ApprovalRequest" type="float"/>
          </mes:decls>
          <mes:template><![CDATA[<l:ApprovalRequest xmlns:l="http://xml.netbeans.org/
              schema/Loans">
            <l:amount>$ApprovalRequest</l:amount>
          </l:ApprovalRequest>]]></mes:template>
        </mes:input>
        <mes:output>
          <mes:decls>
            <mes:typedef name="TApprovalResponse" type="string" values="true,false"/>
            <mes:variable name="ApprovalResponse" type="TApprovalResponse"/>
          </mes:decls>
          <mes:template><![CDATA[<l:ApprovalResponse xmlns:l="http://xml.netbeans.org/
              schema/Loans">
            <l:accept>$ApprovalResponse</l:accept>
          </l:ApprovalResponse>]]></mes:template>
        </mes:output>
      </mes:operation>
    </mes:port>
  </mes:service>
</mes:services>
```

Este catálogo es el resultado de ejecutar ServiceAnalyzer al servicio aprobador del ejemplo, ya explicado anteriormente, del préstamo bancario. Recordemos que el aprobador es el servicio que interviene en la composición cuando el cliente solicita un préstamo muy alto o bajo pero con un riesgo alto.

En el catálogo podemos observar el nombre del servicio, `LoanServiceService`, el puerto[1] y las operaciones (en este caso solo hay una, llamada `grantLoan`) que tienen dos tipos de mensajes: de entrada o salida.

En la definición de estos mensajes se diferencian dos secciones:

[1] El puerto especifica una dirección para el enlace definiendo un único punto de destino de la comunicación.

- Sección de las declaraciones. Donde se definen las variables y sus tipos con sus restricciones.

- Sección de las plantillas. Dependiendo del tipo de las variables se genera una plantilla Velocity que posteriormente se incrustará en el *bpts*.

En el análisis de la composición habrá que tener en cuenta el sentido de los mensajes de las operaciones para poder extraer la plantilla adecuada.

En el *bpts* resultante quedaría algo parecido al listado 5.2.

Listado 5.2: Ejemplo de un *bpts* utilizando plantillas Velocity

```
1  ...
2  <tes:clientTrack>
3    <tes:sendReceive service="loan1:LoanServiceService" port="LoanServicePort" operation
         ="grantLoan">
4      <tes:send fault="false">
5        <tes:template><![CDATA[<l:ApprovalRequest xmlns:l="http://xml.netbeans.org/schema
             /Loans">
6          <l:amount>$ApprovalRequest</l:amount>
7        </l:ApprovalRequest>]]></tes:template>
8      </tes:send>
9      <tes:receive fault="false"/>
10   </tes:sendReceive>
11 </tes:clientTrack>
12 ...
```

Analizador de los conjuntos de correlación

Para poder tratar fácilmente con los conjuntos de correlación, fue necesario implementar otro modelo de datos para poder manejar la información necesaria de esta propiedad. Se implementó en la clase PROPERTYALIASESUTIL.

A medida que se iban analizando las actividades, se comprobaban si éstas formaban parte de un conjunto de correlación. Si era así, se analizaban los ficheros WSDL donde se define esta propiedad y se extraía la información que nos interesa tener. Lo más importante era tener un listado de las partes de todos los mensajes que debían coincidir con un mismo ID y se elegía un ID concreto para todas estas partes.

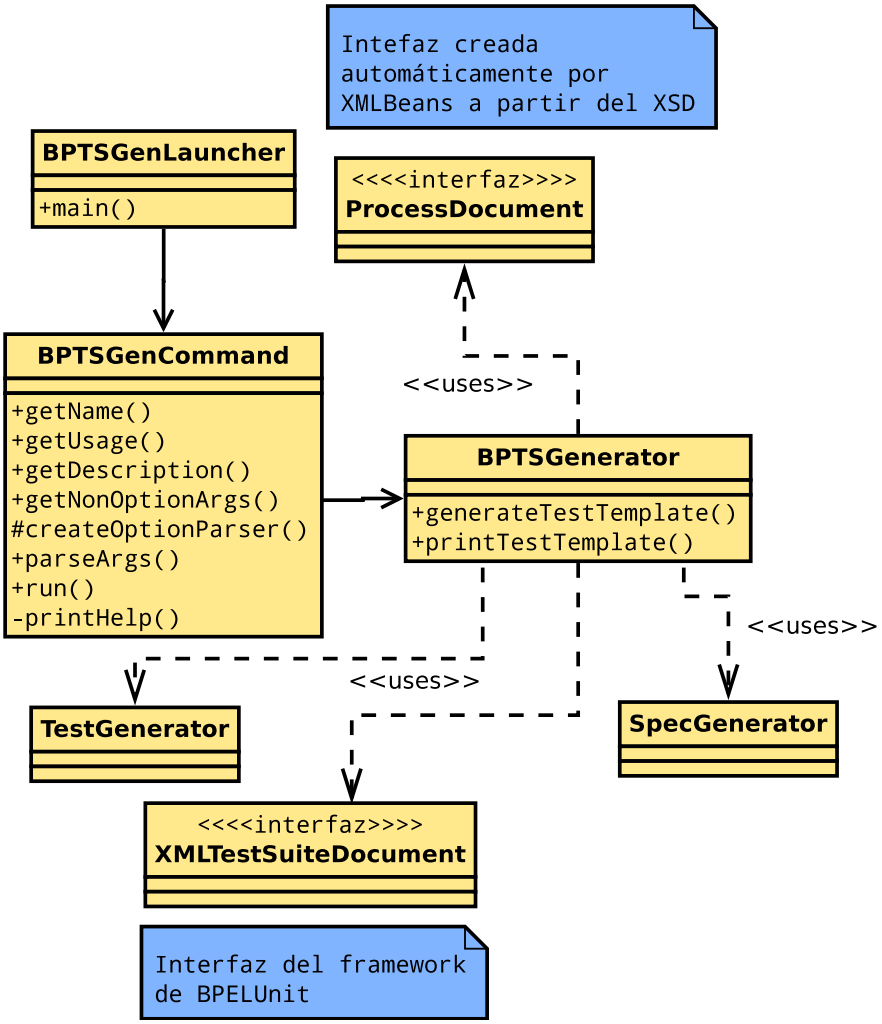

Figura 5.3: Diagrama del generador

5.1.2. Generador

Este bloque es el que da soporte a la generación del fichero *bpts*, unifica y genera el *spec* y por último genera el *vm*.

En la figura 5.3 se presenta el diagrama de clases del generador.

En primer lugar cabe destacar que el manejo de la línea de órdenes y la interfaz del usuario se han separado en las clases BPTSGENLAUNCHER y BPTSGENCOMMAND. Por una parte, se decidió alojar el programa principal que lee los argumentos de la línea de órdenes en la clase BPTSGENLAUNCHER y por otra parte, en BPTSGENCOMMAND se

analizan los argumentos, se muestra la ayuda, etc. Esta división facilita la reutilización del código, la detección de errores, es fácilmente mantenible y es más fiable que ponerlo todo en una única clase.

BPTSGENERATOR es el bloque generador y es a su vez la clase principal del proyecto. Genera la plantilla resultante (el *bpts*) y la muestra por pantalla. Ésta se servirá de la interfaz PROCESSDOCUMENT, que se explicará más adelante, para poder trabajar con la composición mediante objetos. Del mismo modo, utiliza la interfaz XMLTESTSUITEDOCUMENT para poder crear el *bpts*. Finalmente, en esta clase es donde se hace uso de SpecGenerator y TestGenerator para crear los respectivos *spec* y *vm*.

Al igual que en el catálogo que generaba ServiceAnalyzer había que tener en cuenta el servicio, la operación y el sentido de la misma para saber qué plantilla había que extraer, para la generación de los *spec* también hubo que tener en cuenta estos de talles.

A continuación se muestra el *bpts* resultante tras analizar la composición del aprobador:

Listado 5.3: *bpts* resultante tras ejecutar BPTSGenerator

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tes:testSuite xmlns:loan="http://enterprise.netbeans.org/bpel/N6_ServicioPrestamo/
       LoanApprovalProcess" xmlns:loan1="http://j2ee.netbeans.org/wsdl/LoanService"
       xmlns:ass="http://j2ee.netbeans.org/wsdl/AssessorService" xmlns:app="http://j2ee.
       netbeans.org/wsdl/ApprovalService" xmlns:tes="http://www.bpelunit.org/schema/
       testSuite">
3    <tes:name>LoanApprovalProcessTest</tes:name>
4    <tes:baseURL>http://localhost:7777/ws</tes:baseURL>
5    <tes:deployment>
6      <tes:put name="LoanApprovalProcess" type="activebpel">
7        <tes:wsdl>LoanService.wsdl</tes:wsdl>
8        <tes:property name="BPRFile">LoanApprovalProcess.bpr</tes:property>
9      </tes:put>
10     <tes:partner name="assessor" wsdl="AssessorService.wsdl"/>
11     <tes:partner name="approver" wsdl="ApprovalService.wsdl"/>
12   </tes:deployment>
13   <tes:setUp>
14     <tes:dataSource type="velocity" src="data.vm">
15       <tes:property name="iteratedVars">ApprovalRequest AssessorResponse
             ApprovalResponse</tes:property>
16     </tes:dataSource>
```

```
17   </tes:setUp>
18   <tes:testCases>
19     <tes:testCase name="MainTemplate" basedOn="" abstract="false" vary="false">
20       <tes:clientTrack>
21         <tes:sendReceive service="loan1:LoanServiceService" port="LoanServicePort"
               operation="grantLoan">
22           <tes:send fault="false">
23             <tes:template><![CDATA[<l:ApprovalRequest xmlns:l="http://xml.netbeans.org/
                   schema/Loans"><l:amount>$ApprovalRequest</l:amount></l:ApprovalRequest>
                   ]]></tes:template>
24           </tes:send>
25           <tes:receive fault="false"/>
26         </tes:sendReceive>
27       </tes:clientTrack>
28       <tes:partnerTrack name="assessor">
29         <tes:receiveSend service="ass:AssessorServiceService" port="AssessorServicePort
               " operation="assessLoan">
30           <tes:send fault="false">
31             <tes:template><![CDATA[<l:AssessorResponse xmlns:l="http://xml.netbeans.org/
                   schema/Loans"><l:risk>$AssessorResponse</l:risk></l:AssessorResponse>]]>
                   </tes:template>
32           </tes:send>
33           <tes:receive fault="false"/>
34         </tes:receiveSend>
35       </tes:partnerTrack>
36       <tes:partnerTrack name="approver">
37         <tes:receiveSend service="app:ApprovalServiceService" port="ApprovalServicePort
               " operation="approveLoan">
38           <tes:send fault="false">
39             <tes:template><![CDATA[<l:ApprovalResponse xmlns:l="http://xml.netbeans.org/
                   schema/Loans"><l:accept>$ApprovalResponse</l:accept></l:ApprovalResponse
                   >]]></tes:template>
40           </tes:send>
41           <tes:receive fault="false"/>
42         </tes:receiveSend>
43       </tes:partnerTrack>
44     </tes:testCase>
45   </tes:testCases>
46 </tes:testSuite>
```

Recordemos que este tipo de ficheros se divide en dos secciones: de despliegue y de casos de prueba 1.6. En este ejemplo la sección de despliegue está comprendida entre las líneas 3 y 12. Esta sección contiene la información para el despliegue y la

especificación de todos los *partners*.

La sección de los casos de prueba comprende desde la línea 13 hasta el final. En la 13 se encuentra el bloque `<setUp>` donde se describen las variables Velocity. Se pueden declarar directamente dentro de este bloque (ver listado 5.4) o, como en este ejemplo, en un fichero aparte (ver listado 5.5).

Listado 5.4: Declaración de las variables Velocity dentro de `<setUp>`

```
1  <tes:setUp>
2   <tes:dataSource type="velocity">
3     <tes:property name="iteratedVars">
4       cantidad aprobador riesgo aceptado
5     </tes:property>
6     <tes:contents>
7  #set($cantidad = [ 150000, 150000, 1500, 1500, 1500, 3000, 6000, 8000, 8000])
8  #set($aprobador = [ 'true', 'false','silent','true','false','silent','false','smart',
       'smart'])
9  #set($riesto = ['silent','silent', 'low','high', 'high', 'smart','smart','smart','
       smart'])
10 #set($aceptado = [ 'true', 'false', 'true','true','false', 'true','false', 'true','
       false'])
11    </tes:contents>
12  </tes:dataSource>
13 </tes:setUp>
```

En el listado 5.5 tenemos un ejemplo del aspecto que tiene el fichero *vm* generado por TestGenerator. Se puede comprobar que el nombre de las variables coinciden con las declaradas en la línea 15 del listado 5.3.

Listado 5.5: Ejemplo de un fichero *vm*

```
1 #set($ApprovalRequest = [-38016770, 68857761, -15721520, 14025613, -51675714])
2 #set($AssessorResponse = ["high", "low", "low", "high", "low"])
3 #set($ApprovalResponse = ["true", "false", "false", "true", "false"])
```

Siguiendo con la sección de casos de prueba del *bpts* generado por BPTSGenerator, tenemos en la línea 18 el bloque principal que contiene los casos de prueba: `<testCases>`. En un fichero *bpts* que no estuviera basado en plantillas, sería necesario un `testCase` distinto por cada caso de prueba que se quisiera realizar a la composición. En estos ficheros se duplica mucho código, ya que la información de los `<partnerTrack>` no cambia, y pueden ser extremadamente grandes en los casos

de que se quisiera crear muchos casos de prueba. Esto no ocurre en los *bpts* basados en plantillas, ya que como podemos observar en el listado 5.3 solo aparece un `<testCase>`, evitando la duplicidad de código y simplificando enormemente el contenido del propio fichero.

Dentro de este único `<testCase>` tenemos los elementos que BPELUnit utiliza para simular los tres servicios que intervienen en la composición:

- El `<clientTrack>` simula el servicio cliente.
- El `<partnerTrack>` de nombre `assessor` simula al servicio asesor.
- El `<partnerTrack>` de nombre `approver` simula al servicio aprobador.

Al ser una composición síncrona (ver apartado WS-BPEL de §1.5) las dos únicas operaciones que se realizarán son las operaciones `<sendReceive>` y `<receiveSend>`. Las primeras se corresponden con los servicios de la composición que han sido llamados desde la actividad `<receive>` (y a veces la actividad `<onMessage>`, ver §4.2.2), en la que se envía una petición y se espera una respuesta. Análogamente, las segundas se corresponden con los servicios que han sido llamados desde la actividad `<invoke>`, en la que se recibe unos datos y se devuelve una respuesta. Este comportamiento de envío de mensajes es precisamente el que se pretende probar.

Dentro de los bloques de las operaciones antes mencionadas tenemos los mensajes que se enviarán y los que se espera recibir. En este ejemplo solo aparecen con contenido los mensajes que se enviarán (bloque `<send>`), estando el bloque `<receive>` vacío. Esto es así ya que como se explicó anteriormente, extraer de la composición el contenido de lo que devuelven los mensajes es muy complejo y sería necesario aplicar razonamiento automático.

El contenido del bloque `<send>` es la plantilla extraída del servicio específico del catálogo que genera ServiceAnalyzer. Están clasificadas por servicio, puerto, operación y tipo de mensaje (entrada o salida) para saber en qué lugar del *bpts* debe incrustarse.

Estas plantillas son construcciones que representan los mensajes a enviar por el cliente o por un *mockup*. Como ya se ha comentado anteriormente, estas plantillas se parametrizan con el uso de variables permitiendo utilizar un mismo caso de prueba (`<testCase>`) para probar el mismo mensaje con diferentes valores.

5.2. Componentes usados de otras herramientas

A continuación, explicaremos los componentes que se han usado de otras herramientas creadas por el grupo UCASE.

5.2.1. Componentes usados de TestGenerator

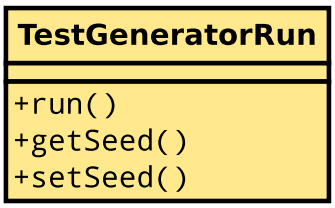

Figura 5.4: Clase TestGeneratorRun de la herramienta TestGenerator

En este caso solamente hizo falta utilizar la clase TESTGENERATORRUN que se encarga de la ejecución del programa.

Los parámetros que hicieron falta especificarle fueron los siguientes:

- El nombre del *spec* (el unificado ya, con todos los tipos y restricciones de las variables de los servicios de la composición).

- El tipo de formato de salida que se desea, ya que TestGenerator trabaja con dos tipos de formato:

  Velocity el cual ya hemos hablado con anterioridad.

  CSV ≪comma-separated values≫ o valores separados por comas en castellano. Este formato describe a un conjunto de formatos de fichero basados en texto plano donde cada línea representa una fila, y los valores de cada fila están separados por un marcador determinado, el cual, normalmente, es una coma.

  En nuestro caso seleccionamos el formato Velocity.

- El número de pruebas que deseamos que tenga el *vm*, el cual por defecto es cinco.

- La estrategia a seguir. Este último valor se deja sin determinar, ya que actualmente el desarrollador de TestGenerator está trabajando en esta labor.

Y por último, mediante la función `run()` da comienzo la ejecución de la herramienta y almacena la salida en un fichero especificado en BPTSCONSTANTS llamado `data.vm`.

5.2.2. Componentes usados de ServiceAnalyzer

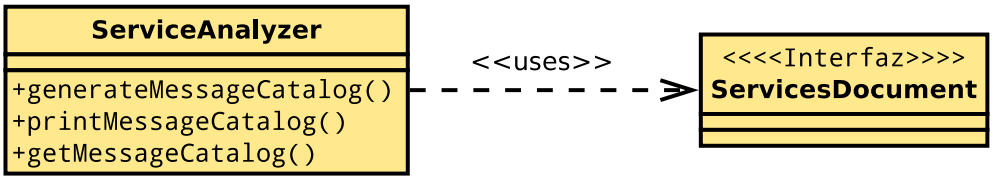

Figura 5.5: Clase ServiceAnalyzer de la herramienta ServiceAnalyzer

En el caso de ServiceAnalyzer, también fue necesaria una instancia de su clase principal, la clase SERVICEANALYZER.

Para conseguir el catálogo solamente hubo que indicarle el fichero WSDL que se deseaba analizar, y ejecutar la función `generateMessageCatalog()`. Sin embargo, para poder manejar este catálogo fue necesario hacer uso de una interfaz de XMLBeans que usa ServiceAnalyzer llamada SERVICESDOCUMENT.

5.2.3. Componentes usados de SpecGenerator

La herramienta SpecGenerator está implementada en una única clase con el mismo nombre SPECGENERATOR.

Los parámetros que necesita para generar un *spec* son los siguientes:

- El nombre del servicio que se esté analizando en un momento determinado.
- La operación de ese servicio.
- Dirección o sentido del mensaje, si es de entrada o salida.
- El catálogo generado por ServiceAnalyzer.

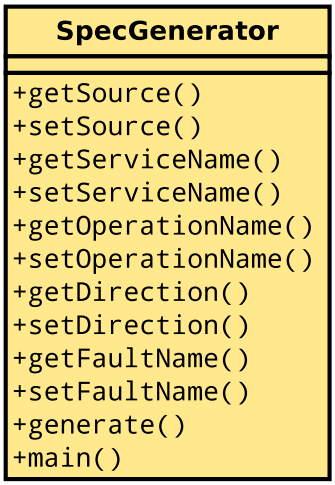

Figura 5.6: Clase SpecGenerator de la herramienta SpecGenerator

Una vez declarados estos valores, se ejecuta una función miembro de SPECGENERATOR llamada `generate()`, que devuelve una cadena que se analizará junto con los *spec* de los otros servicios de la composición para evitar duplicidad de tipos y variables.

5.3. Otras clases utilizadas

5.3.1. Clase BPELPROCESSDEFINITION

Esta clase está integrada en el proyecto bpel-packager, el cual fue creado por el grupo UCASE y está escrito en Java. Se creó con la idea de poder manejar cómodamente información relacionada con el entorno WS-BPEL y sus pruebas con BPELUnit.

Al principio se implementaron las utilidades básicas para los primeros proyectos del grupo, y se ha ido completando con la creación de nuevos proyectos y, consecuentemente, nuevas necesidades.

En concreto, la clase BPELPROCESSDEFINITION se ha utilizado para analizar la composición WS-BPEL, es decir, almacenar toda la información que contiene el fichero WS-BPEL en objetos fáciles de manejar. WS-BPEL está basado en XML, luego para poder navegar por el fichero haría falta un DOM o bien con XPath. Gracias a BPELPROCESSDEFINITION tenemos objetos Java con toda la información a nuestra disposición, sin complicar de manera considerable la codificación.

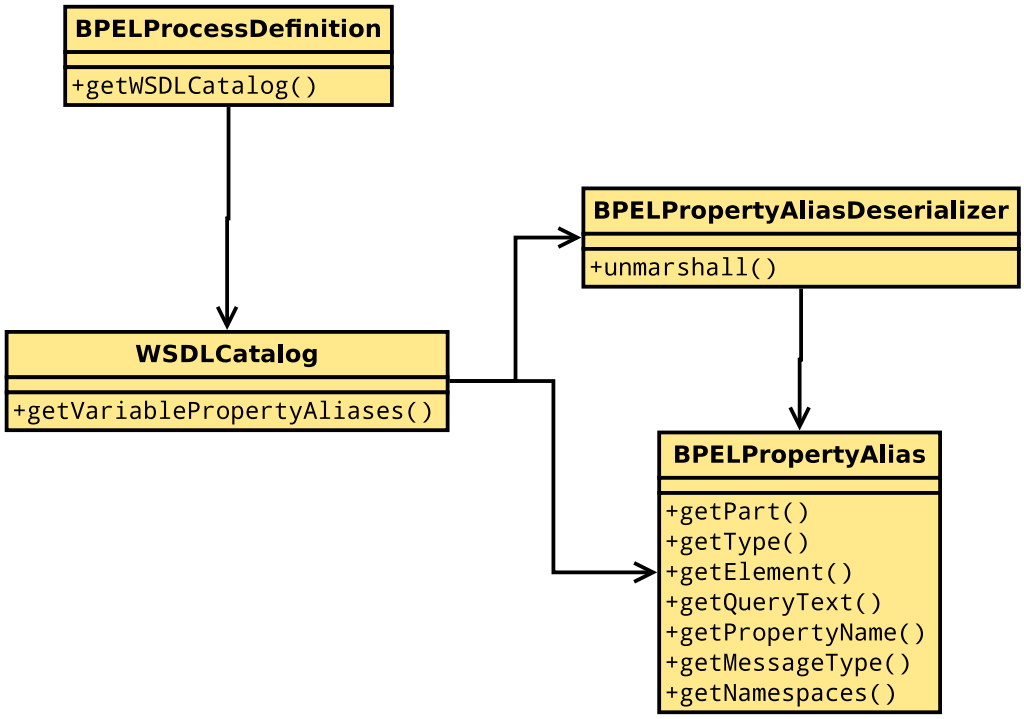

Figura 5.7: Clases de bpel-packager

Además de BPELPROCESSDEFINITION, se han utilizado dos clases más, integradas también en bpel-packager, para poder manejar los conjuntos de correlación de las composiciones WS-BPEL. Estas clases son WSDLCATALOG y BPELPROPERTYALIAS. Fueron necesarios unos pequeños retoques en ambas (además de una tercera llamada BPELPROPERTYALIASDESERIALIZER que extrae, interpreta y almacena información del fichero WSDL), ya que hasta ahora no había sido necesario analizar la parte `<query>` del fichero WSDL, necesaria para la fase de análisis de la composición en la que se analizan estos conjuntos de correlación.

5.3.2. Interfaz PROCESSDOCUMENT

Esta interfaz es generada por XMLBeans a partir de la especificación oficial en XML Schema del formato de los ficheros WS-BPEL. Al igual que BPELPROCESSDEFINITION, extrae la información del fichero WS-BPEL para convertirlos en objetos Java fácilmente manejables. Esta biblioteca provee considerablemente muchas más funcionalidades

que BPELPROCESSDEFINITION, y además, esta herramienta ya se había utilizado en otros proyectos del grupo UCASE, dando buenos resultados, y en vez de añadir funcionalidades a *bpel-packager*, se decidió usar XMLBeans para analizar los datos de la composición.

Con PROCESSDOCUMENT obtenemos en un objeto Java el contenido del WS-BPEL, y con el resto de clases implementadas en XMLBeans conseguimos acceder a cualquier parte de la composición.

5.3.3. Interfaz XMLTESTSUITEDOCUMENT

Esta clase también es generada por XMLBeans, sin embargo, las clases generadas se incluyen en las distribuciones binarias de BPELUnit. Utilizamos el framework de BPELUnit [18] para crear el *bpts*. Para empezar, XMLTESTSUITEDOCUMENT posee una fábrica o `Factory` con el que podemos crear una nueva instancia de la futura plantilla *bpts*. Con el resto de clases del framework iremos completando la plantilla hasta conseguir un *bpts* completo.

Capítulo 6

Implementación y pruebas

En este capítulo hablaremos de las herramientas y tecnologías que se han elegido para el desarrollo de la solución. Explicaremos también el tipo de pruebas realizadas para verificar la validez de la solución.

6.1. Herramientas usadas

6.1.1. Java

Como ya se ha comentado anteriormente, el lenguaje de programación utilizado para implementar el sistema ha sido Java [8]. Es el lenguaje que se ha usado por el grupo UCASE para el desarrollo de todas las herramientas relacionadas con la investigación de las pruebas de mutación. También es como está implementado BPELUnit y las bibliotecas de XMLBeans.

Java es un lenguaje de programación de alto nivel orientado a objetos, desarrollado por un grupo de trece personas dirigido por James Gosling, para la empresa Sun Microsystems en el año 1991. Es un modelo de objetos más simple que C++, Cobol o Visual Basic. Elimina herramientas de bajo nivel, que suelen inducir a errores, como la manipulación directa de punteros o memoria, la cual es gestionada mediante un recolector de basura.

Los objetivos principales de este lenguaje son [20]:

- Cercanía de sus conceptos a los del mundo real.

- Proceso de desarrollo más sencillo y rápido.
- Facilita reutilización de diseño y códigos.
- Modificaciones, extensiones y adaptaciones más sencillas.
- Sistemas más estables y robustos.

Java incorpora una utilidad que ayuda a generar una buena documentación de manera fácil y sencilla, llamada Javadoc [23]. Es una herramienta ofrecida por Oracle para generar documentación de APIs en formato HTML a partir de código fuente Java.

6.1.2. Eclipse

Eclipse [12] es un entorno de desarrollo integrado o IDE (integrated development environment) de código abierto multiplataforma, para hacer lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones de "Cliente liviano" basadas en navegadores.

El proyecto Eclipse fue originalmente creado por IBM en noviembre del 2001 y soportado por un consorcio de empresas software. En la actualidad este proyecto es desarrollado por la Fundación Eclipse. Esta fundación fue creada en enero de 2004 como una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios.

El IDE Eclipse emplea módulos o *plugins* para proporcionar toda su funcionalidad al frente de la plataforma de cliente enriquecido, a diferencia de los entornos monolíticos, donde todas las funcionalidades están incluidas, sean útiles o no para el usuario. Con estos módulos podemos extender la funcionalidad de Eclipse a otras tecnologías como C/C++, Python, LaTeX, aplicaciones en red, sistemas de gestión de bases de datos, etc.

El editor de texto del que dispone Eclipse resalta la sintaxis, compila en tiempo real, pruebas unitarias con JUnit, control de versiones con CVS, integración con Ant, asistentes para la creación de proyectos, clases, tests, etc.

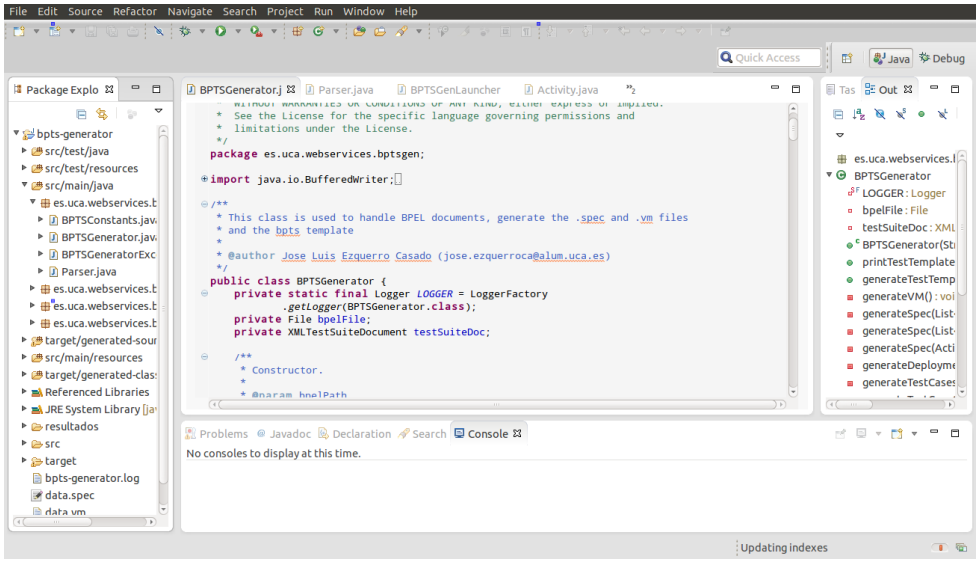

Figura 6.1: Captura de pantalla del IDE Eclipse

6.1.3. Maven

Maven [11] es una herramienta diseñada para la gestión y construcción de proyectos Java. Automatiza la gestión del software, incluyendo compilación, pruebas y despliegue, entre otras tareas. Fue creada por Jason van Zyl, de Sonatype dentro del proyecto Jakarta (2002), aunque actualmente el proyecto pertenece a la Apache Software Foundation.

Su funcionalidad es parecida a la de Apache Ant, también para proyectos Java. La diferencia principal entre ambas es que en Ant las acciones a realizar se definen en forma procedural paso a paso, mientras que con Maven se declara qué *plugins* se van a utilizar, con qué configuración y con qué dependencias y Maven se encarga del orden en el que se utilizan las cosas para lograr el objetivo declarado.

Los objetivos de Maven son las unidades mínimas de ejecución de las que disponemos durante su uso 6.2. Un grupo de objetivos conforman un plugin. La ejecución de un objetivo se dispara desde línea de comandos invocando Maven con el nombre del *plugin* que lo contiene.

Utiliza un POM (Project Object Model) para describir el proyecto software a construir, sus dependencias con otros módulos y componentes externos, así como el orden

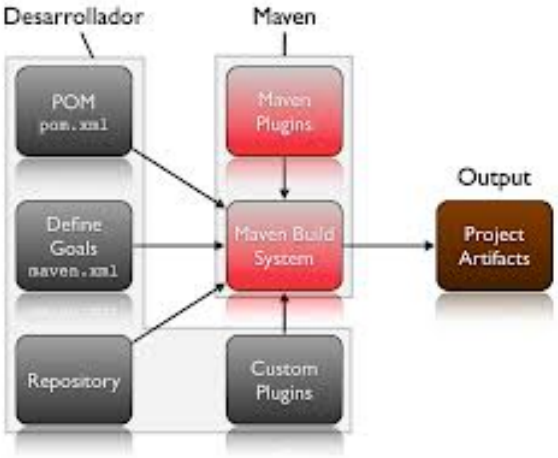

Figura 6.2: Arquitectura de Maven

de construcción de los elementos. Es un fichero basado en XML que se tiene que ubicar en la raíz del proyecto o módulo.

Maven está listo para usar en red, ya que su motor puede descargar dinámicamente los *plugins* de un repositorio. Dicho repositorio provee acceso a muchas versiones de diferentes proyectos como Open Source en Java, de Apache y de otras organizaciones. Este repositorio y su sucesor reorganizado, Maven 2, intentan ser el mecanismo por defecto de distribución de aplicaciones en Java, pero su adopción está siendo lenta. Maven permite subir artefactos al repositorio al final de la construcción de la aplicación, de forma que cualquier usuario tiene acceso a ella.

Usa una arquitectura basada en *plugins* que permite que utilice cualquier aplicación controlable a través de la entrada estándar. En teoría, esto permite que cualquiera pueda escribir sus propios *plugins* para su interfaz con herramientas como compiladores, herramientas de pruebas unitarias, etc. para cualquier otro lenguaje; pero en la realidad, Maven apenas soporta otros lenguajes distintos a Java.

Está construido alrededor de la idea de reutilización de la lógica de construcción: los proyectos se construyen generalmente con patrones similares; una elección lógica sería reutilizar los procesos de construcción. La idea no es reutilizar el código o funcionalidad, sino cambiar la configuración del código escrito. Los proyectos en Maven cuentan con una serie de etapas, llamadas ciclo de vida. Para pasar a una etapa, es ne-

cesario haber completado con éxito las etapas anteriores. Estas etapas representan las distintas fases por las que un proyecto software ha de pasar. Las etapas más importantes son:

compile compila el código.

test ejecuta las pruebas unitarias.

package empaqueta el bytecode resultado de compilar en un .jar.

install instala el .jar en el repositorio local de Maven.

deploy despliega el .jar en el repositorio remoto de Maven.

Otro aspecto interesante de Maven es que, a partir de la información del POM y del código fuente, proporciona al usuario información de los proyectos que puede llegar a serle muy útil: arboles de dependencias, listas de direcciones, informes de las pruebas, referencias cruzadas entre las fuentes, etc. Asimismo, proporciona guías de buenas prácticas para el desarrollador.

6.2. Integración continua

En la figura 6.3 se muestra el proceso que utiliza el grupo UCASE para desarrollar y gestionar las herramientas creadas y sus respectivas pruebas.

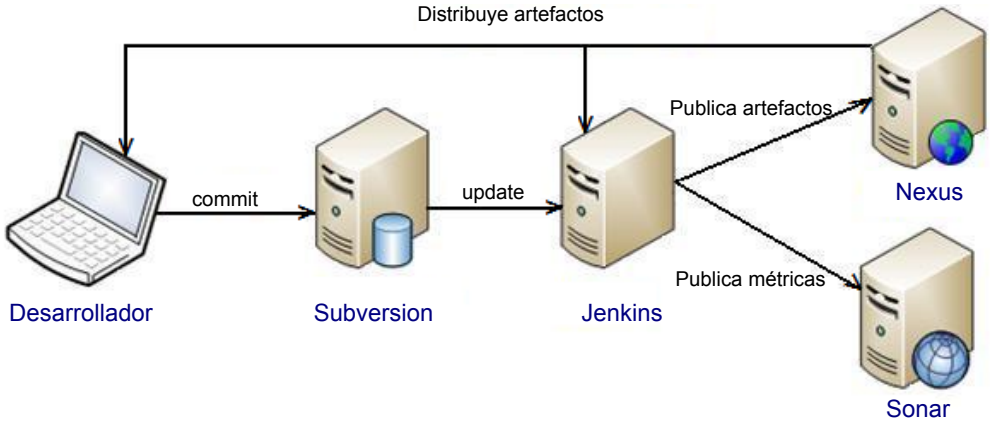

Figura 6.3: Sistema de integración continua

6.2.1. Subversion

Subversion [2] es un sistema de control de versiones. Fue diseñado para reemplazar a CVS. Es un software libre con licencia Apache/BSD. También se le conoce por svn, ya que este es el nombre que usa en la línea de órdenes. Una de las principales diferencias entre Subversion y CVS es que en el primero todo el proyecto está bajo el mismo número de revisión, mientras que en el segundo cada archivo tiene un número de revisión independiente.

El hecho de que se pueda acceder al repositorio a través de la red permite que varias personas trabajen sobre el mismo repositorio. Esto fomenta la colaboración, y la calidad no se resiente, ya que si algún cambio es incorrecto, siempre se puede volver a una versión anterior a dicho cambio

Entre las ventajas existentes de usar Subversión podemos destacar:

- Mantenemos la historia de los archivos y directorios, incluso después de realizar copias y renombramientos.
- Todo los cambios subidos de una sola vez a la forja cuentan como una única modificación, aunque afecte a varios archivos. Esto facilita la vida del programador ya que no tiene que subir archivos uno a uno.
- Al contrario que en CVS, sólo se mandan los cambios que han sufrido los ficheros, y no el fichero completo.
- Maneja los ficheros binarios de forma eficiente.
- Permite el bloqueo de archivos, para que no sean editados por más de una persona al mismo tiempo.
- Cuando se integra con Apache, se puede usar todas las opciones que este servidor posee para autentificar archivos.

El contenido del prestente PFC, está subido a la forja del grupo UCASE usando Subversion. Se elige este sistema por el bajo coste que implica aprenderlo para todos los miembros del grupo, además de por ser el más extendido en todo tipo de proyectos en la actualidad.

6.2.2. Jenkins

Jenkins [13] es un software de integración continua de código abierto escrito en Java, basado en el proyecto Hudson.

Hudson fue creada por Kôsuke Kawaguchi(empleado de Sun) en su tiempo libre, que se encarga de monitorear la ejecución de tareas repetidas, tales como la creación de un proyecto o la ejecución de tareas.

Jenkins descarga, compila y ejecuta el proyecto y todos sus pruebas con la periocidad que se le indique con el objetivo de comprobar que todo funciona como debe.

Algunas de las características a destacar de Jenkins son:

- Es fácil de instalar.
- Monitoriza la ejecución de las tareas.
- Todas las tareas de administración se realizan usando una interfaz web, lo cual facilita su configuración
- Soporta notificación vía IM (Instant Messaging), e-mail y RSS (Really Simple Syndication).
- Genera informes para JUnit.
- La herramienta puede extenderse y personalizarse fácilmente mediante *plugins*.
- Soporta CVS, Git y Subversion para el control de cambios.

En el caso del grupo UCASE se realizarán estas pruebas siempre justo cuando se hayan subido los nuevos cambios y una vez al día haya cambios nuevos o no. De esta forma nos aseguramos que el proyecto siempre funcione de manera correcta. Además, se puede configurar también para enviar correos a los desarrolladores cuando algo no funcione como debiera para que sea solucionado cuanto antes.

6.2.3. Nexus

La distribución de Maven por defecto se descarga los artefactos del repositorio principal de Maven. Si bien esto a nivel personal es factible, cuando varios desarrolladores

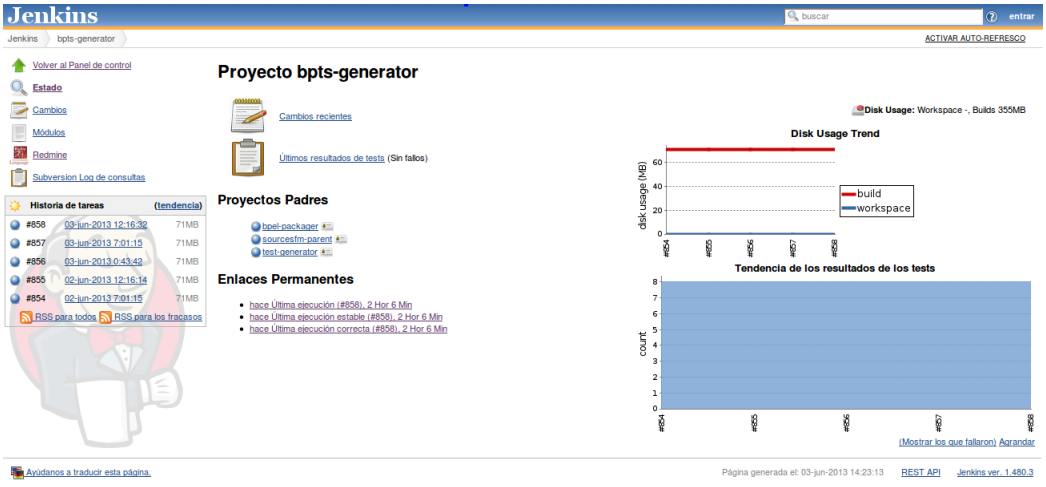

Figura 6.4: Vistazo del proyecto BPTSGenerator en Jenkins

se tienen que descargar los mismos .jars pesados una y otra vez es un despilfarro de ancho de banda y de tiempo considerable. Por otro lado, a una organización podría interesarle controlar o restringir de algún modo los artefactos que pueden descargarse los desarrolladores (para que descarguen una misma versión de una biblioteca, para que no usen el repositorio con fines ajenos a la organización, para que sólo empleen dependencias con una licencia compatible a la del proyecto en el que trabajan, etc.). Por ello se suele instalar un gestor repositorio propio, siendo Nexus una de las mejores opciones. Entre los motivos por los que debe elegirse Nexus, podemos destacar:

- Puesto que está creado por desarrolladores de Maven, la compatibilidad con esta herramienta y la eficiencia en las comunicaciones con su repositorio central son las máximas posibles.
- Es pionero en el formato de repositorio índice.
- Su configuración y mantenimiento son sencillos. Sin embargo, puede ser usado de manera profesional y permite su extensión mediante *plugins*.
- Destaca por poseer el modelo de seguridad más robusto y configurable de entre los gestores de repositorios actuales.

- Nexus usa Apache Lucene para indexar y buscar en tiempo real, sin necesidad de tener repositorios de contenido o bases de datos.

6.2.4. Sonar

Sonar [26] es una herramienta que nos permite controlar la calidad del código. Realiza varios análisis del código usando diferentes herramientas y nos informa de métricas sobre nuestro proyecto que nos serán de utilidad para saber los puntos débiles del mismo. A través de su interfaz intuitiva, nos presenta de forma unificada multitud de métricas: porcentaje de comentarios y de líneas duplicadas, complejidad de métodos y clases, porcentaje de cobertura con las pruebas unitarias así como los puntos para los que no se ha diseñado ninguna prueba...

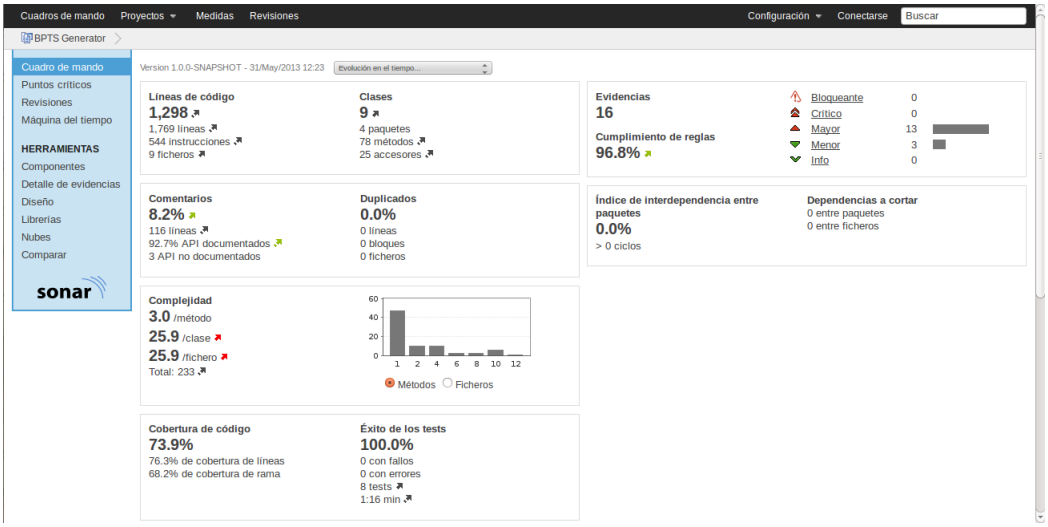

Figura 6.5: Vistazo del proyecto BPTSGenerator en Sonar

Es decir, Sonar permite gestionar la calidad del código controlando los siete ejes principales de dicha calidad del código:

- Arquitectura y diseño.
- Duplicaciones.
- Pruebas unitarias.

- Complejidad.
- Errores potenciales.
- Reglas de codificación.
- Comentarios

En la figura 6.5 podemos ver de forma detallada los puntos débiles del proyecto, antes descritos. Lo primero que llama la atención es la sección de “Evidencias” que nos indica los errores que tiene nuestro código dividido en niveles de gravedad. Ésta es una visión muy útil para asegurar que nuestro código está escrito de acuerdo a las buenas prácticas de Java mejorando así en eficiencia, usabilidad y mantenibilidad, fundamentalmente.

Esta pantalla también da información del resultado de los test que en nuestro caso pasa el 100 % de ellos, y de su cobertura. La cobertura de código es un dato muy útil porque nos muestra las partes que no se han realizado pruebas.

Muestra información sobre el porcentaje de líneas que son comentarios y de líneas duplicadas en el código. Este último dato nos puede servir para darnos cuenta de las zonas de la aplicación que están repetidas y que convendría refactorizar en una única clase, aunque en nuestro caso no es así ya que está al 0 %.

Entre las correcciones que se han realizado al proyecto gracias a la información proporcionada por Sonar podemos destacar:

- Corte de relaciones cíclicas: había varios módulos con un índice de complejidad ciclomática muy elevado que se redujo considerablemente.
- Excepciones redundantes que se declaraban en algunas funciones al añadir código.
- Métodos y atributos que no se usaban en el proyecto.
- Salidas por pantalla.

6.3. Pruebas

En el Capítulo 3 de la presente memoria se han explicado los conceptos sobre las pruebas del software y la importancia de las mismas. A continuación se detallarán las pruebas que se han realizado al código del proyecto.

6.3.1. Metodología de las pruebas

Alcance

Como ya se ha comentado en repetidas ocasiones, este Proyecto tendrá continuidad dentro del grupo UCASE, por lo que se ha intentado automatizar el mayor número posible de pruebas para facilitar el trabajo en caso de producirse modificaciones o ampliaciones en un futuro.

Tiempo y lugar

Las pruebas se han llevado a cabo a lo largo del desarrollo del proyecto. Con el estudio más exhaustivo de la lógica de negocio de las composiciones WS-BPEL, se iban añadiendo al apartado de pruebas composiciones más complejas que incorporaban alguna característica más a este estudio.

Naturaleza de las pruebas

La totalidad de las pruebas realizadas son pruebas de caja negra.

Recordemos que las pruebas de caja negra son aquellas que se centran en lo que se espera de un módulo, intentan encontrar casos en los que el módulo no atiende a la especificación, al contrario que las pruebas de caja blanca que se centran en comprobar que la estructura interna del software es la adecuada

El problema con las pruebas de caja negra es que el conjunto de los datos posibles a probar donde no se cumple la especificación es demasiado extenso. Para combatirlo, se sigue una técnica algebraica conocida como clases de equivalencia. Consiste en dividir el rango de valores permitidos en diferentes clase. Dentro de una misma clase la salida debe ser la misma obligatoriamente, y se debe probar al menos un valor de cada clase.

De esta forma aseguramos, probando un valor de cada clase, que para todo el rango de valores se cumple la salida esperada.

Los casos de prueba se han implementado usando el framework JUnit, un conjunto de bibliotecas creadas para hacer pruebas en aplicaciones Java.

6.3.2. Plan de pruebas

Como se ha comentado anteriormente, las pruebas del proyecto se han centrado en el éxito de la creación del *bpts*, probando composiciones con las que trabaja el grupo UCASE.

Batería de pruebas ABSTRACTCOMPOSITIONTEST

De esta clase heredan todas las clases de prueba que se han implementado. En ella se prueban aspectos comunes a todas tales como validar el *bpts*[1], el URL correcto y el nombre no nulo del documento. También se especifican funciones que utilizarán las clases de prueba heredadas, necesarias para comprobar más información:

- El nombre del fichero WSDL debe ser el que corresponda con la composición.
- El nombre del servicio del `<clientTrack>` debe ser el mismo que el cliente de la composición.
- El número de `<partnerTrack>` que debe tener la plantilla.
- El número de *partners* definidos en las pruebas tiene que ser el mismo que en la composición.

La generación del *bpts* se ha probado con ejemplos de algunas composiciones con las que trabaja el grupo UCASE mediante las siguientes clases de prueba:

- LOANAPPROVALDOCTEST: La composición *Loan Approval* representa un servicio de préstamo bancario. Se han realizado un total de 7 pruebas con una duración de ejecución de 6.1 segundos.

[1] Gracias a la clase TESTSUITEXMLVALIDATOR que comprueba que el *bpts* creado reúne los requisitos descritos por BPELUnit.

- LOANAPPROVALRPCTEST: Es una modificación de la composición anterior con llamadas a procedimientos remotos. Se han realizado un total de 7 pruebas con una duración de ejecución de 5.6 segundos.

- MARKETPLACETEST: Es una composición que trabaja con dos peticiones: la del comprador y la del vendedor. A parte de las pruebas comunes realizadas al resto de composiciones se añade una para comprobar el ID correcto de los mensajes ya que esta composición utiliza los conjuntos de correlación (ver §4.2.3). Se han realizado un total de 8 pruebas con una duración de ejecución de 5.6 segundos.

- MARKETPLACEFLOWTEST: Esta composición tiene la misma funcionalidad que la anterior cambiando únicamente la manera en la que se ha implementado, donde en vez de utilizar actividades `<receive>` utiliza mensajes `<onMessage>`. Se han realizado un total de 8 pruebas con una duración de ejecución de 8 segundos.

- METASEARCHTEST: Es un servicio de un motor de búsquedas por internet. Se han realizado un total de 7 pruebas con una duración de ejecución de 30.8 segundos.

- SHIPPINGSYNCTEST: Implementa un servicio de envío de paquetes. Se han realizado un total de 7 pruebas con una duración de ejecución de 5.9 segundos.

- SQUARESSUMTEST: Este ejemplo básicamente realiza un sumatorio de variables al cuadrado $\sum_i^n i^2$. Se han realizado un total de 7 pruebas con una duración de ejecución de 26.8 segundos.

- TACSERVICETEST: Es una composición muy sencilla que invierte las líneas que se le envían. Se han realizado un total de 7 pruebas con una duración de ejecución de 6.7 segundos.

Por último, se han ejecutado las pruebas generadas sobre la composición, para comprobar que las pasa correctamente.

6.4. Otras herramientas

6.4.1. LaTeX

LaTeX [16] s un sistema de composición de textos, orientado a la creación de textos científicos y técnicos en especial.

LaTeX está formado por un gran conjunto de macros de TeX, escrito por Leslie Lamport en 1984, con la intención de facilitar el uso del lenguaje de composición tipográfica, TeX.

LaTeXse extendió rápidamente por todo el sector científico y técnico gracias a su facilidad de uso y toda la potencia de TeX. Su código abierto permitió que muchos usuarios realizasen nuevas utilidades que extendiesen sus capacidades con objetivos muy variados, apareciendo "dialectos" de LaTeX, muchas veces incompatibles entre sí. En 1993 se anunció una reestandarización completa de LaTeX para evitar discrepancias anteriores, creándose nuevas extensiones como la posibilidad de escribir transparencias por ejemplo.

La característica más relevante que se creó fue la arquitectura modular. Se estable un núcleo central, el compilador, que mantiene las funcionalidades de la versión anterior pero permite incrementar su potencia y versatilidad por medio de diferentes paquetes, que cualquiera puede crear uno nuevo, que sólo se cargan si son necesarios.

La totalidad de esta memoria está desarrollada con LaTeX. Además del diseño limpio y claro que proporciona al documento de manera automática, crea índices, referencias, bibliografía, ajusta figuras y listados... y un sinfín de características que facilitan enormemente la labor generando además un documento impecable, que difícilmente se conseguiría con otro tipo de procesadores.

6.4.2. Dia

Dia es una aplicación informática de propósito general para la creación de diagramas, desarrollada como parte del proyecto GNOME. Está concebido de forma modular, con diferentes paquetes de formas para diferentes necesidades.

Dia está diseñado como un sustituto de la aplicación comercial Visio de Microsoft. Se puede utilizar para dibujar diferentes tipos de diagramas. Actualmente se incluyen

diagramas entidad-relación, diagramas UML, diagramas de flujo, diagramas de redes, diagramas de circuitos eléctricos, etc. Nuevas formas pueden ser fácilmente agregadas, dibujándolas con un subconjunto de SVG e incluyéndolas en un archivo XML.

El formato para leer y almacenar gráficos es XML (comprimido con gzip, para ahorrar espacio). Puede producir salida en los formatos EPS, SVG y PNG, entre otros.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Valoración personal

La elaboración de este proyecto ha exigido un gran aprendizaje y esfuerzo por parte del alumno. Ha sido el proyecto de mayor envergadura realizado durante toda la carrera. Realizar un trabajo tan complejo y de tanta duración ha supuesto un cambio importante a lo que venía realizando los años anteriores en la universidad.

La experiencia de colaborar en un grupo de investigación ha sido muy enriquecedora, tanto por el propio proyecto como los seminarios semanales elaborados por el grupo. Esto ha facilitado aumentar las competencias transversales del alumno como son el trabajo en equipo, la expresión oral y escrita así como intervenciones en público.

Trabajar en un entorno de integración continua con dependencias entre varios proyectos pone en práctica muchos de los conceptos teóricos y buenas prácticas de programación hasta ahora sólo estudiadas en libros. El uso de Subversion, supone valorar las ventajas que proporciona el uso de herramientas de control de versiones y de Sonar, una herramienta tremendamente útil, el tener un código de calidad y la manera de lograrlo.

La mayoría de las tecnologías necesarias para la realización de este proyecto no habían sido aprendidas hasta ahora, con el añadido de que algunas de ellas están en fase de desarrollo, con lo que la documentación es escasa y los cambios continuos.

Ya había trabajado antes con el lenguaje de programación Java, pero no se había

estudiado en profundidad ni se sabía de todo su potencial y facilidad de programación.

Por último, decir que se ha realizado una labor de documentación, tanto manuales como *javadocs*, y se ha decidido que el proyecto sea Software Libre, siguiendo con la misma línea de los proyectos del grupo UCASE y para que pueda resultar de provecho para la comunidad de desarrolladores.

7.2. Tabajo futuro

Aunque el alumno termina sus estudios, este proyecto va a tener continuidad dentro del marco de trabajo del grupo UCASE, ya que es de gran utilidad para probar las nuevas composiciones WS-BPEL que se desarrollen.

Principalmente se mejorará el estudio de la lógica de negocio de las composiciones, tal y como se comentó en el capítulo 4, puesto que un estudio más exhaustivo conlleva el conocer todos los posibles caminos de una composición: estudiar las invocaciones de los servicios, analizar los mensajes que se envían, e incluso conocer el contenido de los mensajes que se reciben para estos datos en el catálogo de pruebas.

Por otra parte, las pruebas no se limitarán a estudiar composiciones síncronas sino que abarcará también el estudio de composiciones asíncronas (ver 1.6).

Finalmente, se estudia la posibilidad de crear un *plugin* para el IDE Eclipse, puesto que ya existen *plugins* para este editor con los que poder trabajar con WS-BPEL, WSDL, e incluso BPELUnit tiene un framework para crear los ficheros *bpts*. Así que lo ideal es mejorar este aspecto para que al crear una composición WS-BPEL con un solo *click* crear un *bpts* con su catálogo de pruebas en el fichero *vm*.

Anexo A

Manual de usuario

En esta manual hablaremos de los pasos necesarios para poder instalar y usar la herramienta.

A.1. Instalación de BPTSGenerator

Para instalar esta herramienta bajo una distribución Linux (para este manual se ha usado Ubuntu 12.04), hay que realizar los siguientes pasos:

1. Hay que acceder al módulo BPTSGenerator de la última versión estable que se ejecutó en Jenkins: `https://neptuno.uca.es/jenkins/job/bpts-generator/lastStableBuild/` A continuación nos descargamos el archivo *uberjar.zip* o *uberjar.tar.gz*.

2. Descomprimimos el archivo en un directorio del sistema que se desee.

3. Añadimos dicho directorio a la variable path del sistema operativo. Por ejemplo, si se ha descomprimido en la carpeta Documentos del usuario, introduciremos la siguiente orden en una terminal:

```
PATH="$PATH:/home/usuario/Documentos/bpts-generator-1.0.0-SNAPSHOT"
```

4. Para que los cambios sean permanentes, hay que modificar el fichero *~/.bashrc*. Para ello abrimos este fichero desde terminal con la herramienta *gedit*, por ejemplo:

```
gedit ~./bashrc
```

   y al final del documento añadimos la siguiente línea:
   `export PATH=$PATH:/home/usuario/Documentos/bpts-generator-1.0.0-SNAPSHOT`

5. Para que los cambios surtan efecto habrá que cerrar sesión y volver a entrar.

A.2. Uso de la herramienta

Para usar la herramienta es necesario abrir una terminal y escribir la siguiente sentencia:

```
bpts−generator (argumentos)
```

Donde "argumentos" puede tomar los siguientes valores:

- `-help`
- `archivo.bpel`

Si recibe el argumento `-help` mostrará la manera de usar la herramienta y una breve descripción de la misma.

Si por el contrario recibe un archivo *bpel* se ejecutará mostrando por pantalla el contenido de la plantilla *bpts*.

Para poder almacenar este *bpts* en un documento se puede desviar el flujo de salida por terminal a un fichero de la siguiente manera:

```
bpts−generator LoanApprovalDoc.bpel > LoanApprovalDoc−plantillaVelocity.bpts
```

Anexo B

Manual del desarrollador

Este manual está orientado a los usuarios que deseen acceder al código fuente del proyecto. Explicaremos los pasos que serán necesarios para descargar, compilar y ejecutar el código fuente, para a partir de ahí, realizar los cambios oportunos.

El presente manual está orientado a usuarios de GNU/Linux, y la distribución que se ha utilizado es Ubuntu 12.04.

B.1. Instalación previa de herramientas

Para obtener el código fuente y poder realizarle cambios es necesario, primeramente, tener conexión a internet, y tener instalado en el equipo Subversion, el JDK de Java, Maven y JUnit. Para instalar Subversion abrimos una terminal y ejecutamos la siguiente orden:

```
sudo apt-get install subversion
```

Y para instalar el resto de tecnologías, en la misma terminal tecleamos:

```
sudo apt-get install openjdk-6-jdk maven junit
```

Se recomienda el uso de un IDE con el que poder desarrollar el proyecto. En particular se recomienda Eclipse. Este IDE se puede descargar de su página web oficial:

`http://www.eclipse.org/`

En la sección de descargas, nos descargaremos la versión que pone *Eclipse IDE for Java EE Developers*. Extraeremos la carpeta que viene en el paquete que se ha descargado y ejecutaremos Eclipse haciendo doble clic en el ejecutable *eclipse*.

B.2. Descarga y preparación del proyecto

Para descargar el código fuente del repositorio, usaremos Subversion. Abriremos una terminal y vamos al directorio donde queremos alojar el proyecto. Luego escribimos la siguiente orden:

```
svn checkout https://neptuno.uca.es/svn/sources-fm/trunk/src/bpts-generator/
```

Se descargarán todos los ficheros obteniendo una copia local de la última versión del proyecto y sus últimas actualizaciones. Ya podemos navegar por el proyecto, para explorar el código nos dirigiremos a la carpeta *src*. El código está organizado de la siguiente manera:

- *main/java*: Directorio principal con el código Java.
- *test/java*: Directorio con el código de las pruebas.
- *main/resources*: Contiene los recursos utilizados por el código principal.
- *test/resources*: Contiene los recursos necesario para ejecutar las pruebas.
- *main/assembly*: Aquí se encuentran los descriptores de distribuciones.

A continuación, se descargarán todas las dependencias de los proyectos usando Maven. A la hora de desarrollar, le serán útiles los objetivos predefinidos de Maven. A continuación se exponen los principales:

clean Elimina todos los directorios de despliegue del proyecto.

compile Compila el código fuente del proyecto.

deploy Despliega el paquete resultante en un repositorio central para ser compartido.

install Instala el paquete en el repositorio local.

site Genera un sitio con la documentación del proyecto.

test Ejecuta las pruebas del proyecto.

Maven nos facilitará enormemente la tarea pues tan sólo con entrar a través del terminal a la carpeta de los proyectos tendremos todo listo una vez hayamos ejecutado:

```
mvn compile
```

Además podemos preparar el proyecto para trabajar con Eclipse escribiendo lo siguiente:

```
mvn eclipse:eclipse
```

Aunque se puede realizar en una misma orden, y si adicionalmente se desean descargar tanto los ficheros fuente de las dependencias como la documentación de las mismas se añadiría lo siguiente:

```
mvn compile eclipse:eclipse -DdownloadJavadocs -DdownloadSources
```

Una vez realizados estos sencillos pasos, sólo nos quedará importar los proyectos dentro de Eclipse:

1. En Eclipse pulsaremos en File $\longrightarrow$ Import.
2. Desplegaremos el menú General seleccionando Existing Projects into Workspace y por último, pulsaremos en Next.
3. En la nueva pantalla, dejamos marcada la opción Select root directory, con el navegador seleccionamos las carpetas que queramos importar y pulsamos en Finish.
4. En el menú principal, pulsamos en Window $\longrightarrow$ Preferences.
5. Desplegamos el panel Java $\longrightarrow$ Build Path $\longrightarrow$ Classpath Variables y pulsamos en New.
6. Nos aparecerá un nuevo diálogo y en el campo Name introduciremos “M2_REPO”; en el campo Path, seleccionaremos el fichero ˜/.m2/repository.

B.3. Ejecución de las pruebas

Una vez nos hemos descargado el proyecto y compilado con `mvn compile`, podemos ejecutar las pruebas desde una terminal con la siguiente orden[1]:

```
mvn test
```

Desde Eclipse se pueden ejecutar una por una las pruebas siguiendo el siguiente orden:

1. En la parte izquierda del editor tenemos una sección de exploración de los proyectos abiertos.
2. Desplegamos el proyecto bpts-generator.
3. Desplegamos la sección `src/test/java`. Aparecerán los distintos ficheros de prueba del proyecto.
4. Abrimos cualquiera de los ficheros que se desea probar.
5. Una vez estamos en el fichero *.java* de prueba, pulsaremos Run $\longrightarrow$ Run As $\longrightarrow$ JUnit Test.

[1] Nota: hay que estar en el directorio ~/bpts-generator/

Bibliografía

[1] L. Bass, P. Clements, y R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2a. edición, 2003.

[2] Ben Collins-Sussman, Brian W. Fitzpatrick, y C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, primera edición, 2004. Disponible gratuitamente en `http://svnbook.red-bean.com/nightly/en/index.html`.

[3] Laboratorio Nacional de Calidad del Software de INTECO. *Ingeniería del Software: Metodologías y ciclos de vida*, 2009.

[4] Edsger W. Dijkstra. *Chapter I: Notes on structured programming*, 1972.

[5] Juan José Domínguez Jiménez, Emma Blanco Muñoz, Antonio García Domínguez, y Inmaculada Medina Bulo. *Propuesta de una arquitectura para la generación de mutantes de orden superior en WS-BPEL*.

[6] Juan José Domínguez Jiménez, Antonia Estero Botaro, Lorena Gutiérrez Madroñal, y Inmaculada Medina Bulo. *Evaluación de la calidad de los mutantes en la prueba de mutaciones*.

[7] Juan José Domínguez Jiménez, Antonia Estero Botaro, Inmaculada Medina Bulo, Manuel Palomo Duarte, y Francisco Palomo Lozano. *El reto de los servicios web para el software libre*. *Proceedings of the FLOSS International Conference*, 2007.

[8] Bruce Eckel. *Piensa en Java*. Pearson Education, 2007.

[9] Apache Software Foundation. *XMLBeans*, diciembre 2009. URL `http://xmlbeans.apache.org/`.

[10] Apache Software Foundation. *Apache Velocity*, noviembre 2010. URL http://velocity.apache.org/engine/devel/.

[11] Apache Software Foundation. *Apache Maven*, marzo 2013. URL http://maven.apache.org/.

[12] Eclipse Foundation. *Eclipse*, 2013. URL http://www.eclipse.org/.

[13] Antonio García Domínguez. *Instancia Jenkins en Neptuno*, 2013. URL http://jenkins-ci.org/.

[14] Cristina Jiménez Gavilán. *Analizador de Servicios Web basados en WSDL 1.1 para pruebas paramétricas*, 2011. URL http://hdl.handle.net/10498/11695.

[15] JUnit.org. *Resources for Test Driven Development*, mayo 2013. URL http://www.junit.org/.

[16] Proyecto LaTeX. URL http://www.latex-project.org/.

[17] Daniel Lübke y Antonio García Domínguez. *TEMPLATES.markdown en la forja de BPELUnit*. URL https://github.com/bpelunit/bpelunit/.

[18] Daniel Lübke y Antonio García Domínguez. *BPELUnit - The Open Source Unit Testing Framework for BPEL*, noviembre 2010. URL http://bpelunit.net/.

[19] Philip Mayer. *Design and Implementation of a Framework for Testing BPEL Compositions*, septiembre 2006.

[20] Inmaculada Medina-Bulo, Gerardo Aburruzaga-García, y Francisco Palomo-Lozano. *Fundamentos de C++. Segunda edición*. Servicio de publicaciones de la Universidad de Cádiz, Cádiz, 2009.

[21] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, USA, 1979. ISBN 0471043281.

[22] OASIS. *Web Services Business Process Execution Language 2.0*, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[23] Oracle. *Página oficial de Javadoc*. URL http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

[24] Miguel Ángel Pérez Montero. *Generador de casos de prueba aleatorio basado en especificaciones abstractas*, 2012. URL http://hdl.handle.net/10498/14661.

[25] Roger S. Pressman. *Ingeniería del Software: Un Enfoque Práctico*. McGraw-Hill, sexta edición, 2005.

[26] SonarSource S.A. *Sonar*, 2013. URL http://www.sonarsource.org/.

[27] M. Shaw y P. Clements. *A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems*. Proc. COMPSAC, ago 1997.

[28] IEEE Computer Society. *Standard for Software Units Testing*. dic 1990.

[29] Asir S. Vedamuthu. *Web Services Description Language (WSDL) Version 2.0 SOAP 1.1 Binding*. jun 2007. URL http://www.w3.org/TR/2007/NOTE-wsdl20-soap11-binding-20070626.

[30] W3C. *Especificación SOAP*. URL http://www.w3.org/TR/soap/.

[31] W3C. *XML Schema*, diciembre 2009. URL http://www.w3.org/XML/Schema.

[32] WS-I. *Basic Profile Version 1.1*, 2006. URL http://www.ws-i.org/Profiles/BasicProfile-1.1.html.

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable

for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered

to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you

include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to

limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder

explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities

for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with . . . Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.